6502 Based Development System
and
Remote Memory Testing System

A report submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in

Electrical and Computer Engineering

by Matthew K. Farrens

University of Wisconsin - Madison

December 19, 1984

TABLE OF CONTENTS

INTRODUCTION

This project was designed to meet a need by the Nuclear Engineering department for a system that would allow the monitoring of the status of static RAM memory chips while they were undergoing irradiation in the University of Wisconsin nuclear reactor. The project consisted of two main parts. First, a development system had to be developed that would allow the fairly inexperienced user to conceive, design and implement different types of programs and hardware circuits without requiring an exorbitant amount of effort. Second, the remote memory chip testing system had to be built and demonstrated to show that the concept of testing while irradiating was a viable one. This report is the description of how these two goals were met.

The first part of this report contains a description of the actual 6502-based development system, what it consists of, and why it is felt that it has met the desired goals. Following that, the remote memory chip testing setup will be described, including all the hardware and software involved in the remote memory chip tests. Finally, an outline of the future of the remote memory chip test project will be presented.

1

THE 6502-BASED DEVELOPMENT SYSTEM

The first goal of the project was to create an environment that would allow the novice user to write programs in assembly language, download them to a microprocessor based system, and easily debug them. A choice of which host microcomputer system and which microprocessor chip to use in the development system had to be made.

The Osborne I microcomputer was chosen as the host system for several reasons. There are two available for use, which means that there is always a spare present in case of any hardware problems. They are very versatile machines, coming complete with a word processor, two programming languages, and the standard CP/M operating system. Many different additional programming languages are available for these machines, one of which was purchased to be used in this project. The decisive advantage of the Osborne I, however, is its portability. This facilitated transportation to and from the nuclear reactor laboratory.

The microprocessor chosen for use in the development system hardware was the Mostek 6502. This is a very popular and available microprocessor chip which is used in both the Apple and Commadore lines of personal computers. It has no great advantage over other microprocessors on the market in terms of speed or flexibility, but it does have two features that made it the best

choice of processing chips for this project. The 6502 has a
memory-mapped I/O architecture. It has been my experience in
teaching about microprocessors that the concept of memory-mapped
I/O is easier for the novice to comprehend than is the more
conceptually complex I/O mapped architecture used by the Intel
family of microprocessors. Since the goal of this project was to
create a system that an inexperienced user can easily use, it was
important to choose a processor that is fairly easy to under-
stand.

The 6502 also handles its READY line in an advantageous
manner. On this microprocessor, when the READY line is brought
low all internal processing ceases. However, the state of the
address lines at the time of the READY line transition is con-
stantly refreshed so that by connecting simple logic to the CPU
and address lines the processor can be stopped with the next
memory address to be fetched latched on the address lines and
available for observation. This ability does not exist in the
Motorola family of processors. They provide means for single-
cycling their processors, but the methods require much more
hardware and are not nearly as instructive.

Once the choice of host and microprocessor had been made,
work could begin on creating a useful system out of them. At
least four things are needed to maximize the usefulness of the
system. They are listed here below and a description of each one
individually is provided in the following sections.

1)   A 6502 Assembler
2)   An Eprom Programmer
3)   A Monitor to provide useful functions to the user
4)   A Single-stepper to allow the stopping of the processor

Further information is contained in the appendices, which includes program listings of all the software written for this project, flow charts of most of the software and user manuals to aid the user in operating the system.

6502 ASSEMBLER

In order to allow assembly language code to be written for the 6502, an assembler was required. Therefore, writing an assembler was the first task to be accomplished.

The 6502 assembler that was written is a standard two pass assembler. During the first pass the symbol table is created and filled, and any errors that are detected are associated with the line in the file in which they occurred. Two output files are also created for use by the second pass.

The second pass checks each input line to see if the line has an error associated with it. If not, it produces the actual machine code for the instruction and updates the two output files. If there is an error, the line in which the error occurred and the error type are written to the screen and to one of the output files, and no code is produced for that line.

The two output files created are a .LST file that contains each line of code and associated machine code values, and a .REL file that contains the assembled 6502 machine code. If there was an error in a line, the line and the error will appear in the output .LST file. The .REL file contains the assembled code in a format ready for downloading to an external source. The .REL file does not contain the assembled code in Intel Hex format but rather a format created for this project.

Labels are limited in length to 8 characters; if there is a label it must begin in the first column, and if there is not a label, the code must begin in some other column. Due to these restrictions, it is not a free-format assembler, but it is of the same type. A small users manual is contained in this report which has more detailed information concerning the use of the assembler.

There were three versions of this assembler written. The first version was written in a language called CBASIC, a language that comes with the Osborne. (CBASIC stands for Compiled BASIC.) The language is quite slow, however, and it is well known that BASIC is not a well-structured language. Therefore, after using the CBASIC version of the assembler for a year, it was decided that a faster and more structured assembler was needed. To help meet this goal, TURBO Pascal was purchased from Borland International. It provided the structured programming language necessary to write an assembler that is structured and regular enough to be readily modified by someone else if errors are discovered or if improvements and/or modifications are desired.

There are some problems associated with the use of microcomputers that do not occur when using a minicomputer that can seriously affect the methods used to accomplish certain tasks. One problem is with processor speed. As mentioned above, the original assembler was rewritten in Pascal because of the inherent structure and regularity of the language. Originally an attempt was made to write the code in purely standard Pascal to

6

allow for complete compatability between systems. However, this was causing such severe performance penalties that the attempt was abandoned and many non-standard commands of Turbo Pascal were incorporated.

Another major problem with microcomputers is limited disk and memory space. The first Pascal version of the assembler was actually the most efficient and used the best algorithm. Each entire line in the assembly language program has to be parsed so that the program counter can be properly incremented. Thus it is advantageous to carry as much information as possible over from the first pass to the second pass to eliminate the need for a second complete parsing of each line. Unfortunately, storage of all of the necessary information proved so memory-intensive that the assembler could not handle extremely long files. While it is true that it is very unlikely that the novice user will write programs long enough to cause the assembler to fail, this is not a restriction that should be placed on him. Therefore, a second Pascal version was written which is somewhat slower but puts virtually no limits on the programmer. In this version the only information carried over between passes is the error status of the line. This requires two complete parses of each line to be performed, but this does not seriously degrade the performance, since the Osborne seems to be mostly disk I/O bound. This is the version that appears in this report. Even this slightly slower version has recorded an improvement in speed of over 400% compared to the original assembler written in CBASIC. When

debugging assembly language programs, constantly having to re-assemble the code provides a definite appreciation of the speed increase.

Since the Osborne does seems to be mainly disk I/O bound, some possible improvements to the assembler include directives that would inhibit the production of the .LST and/or the .REL files, and perhaps the ability to prevent the printing of the symbol table. The way the code is written, these kinds of modifications should not be extremely difficult to perform. Any reduction in the amount of data written out to the disk should significantly increase the performance of the program.

6502 MONITOR

The 6502 development system needs to provide a means for downloading test programs and provisions for debugging the downloaded software. This capability is provided by the monitor program, in conjunction with the Single-Stepper.

The monitor program was modeled loosely after the monitors that are used by the students in ECE 453. In the current development system it resides in memory from address F800 to FBFC hex. Conceptually it would be best if the monitor existed in the highest block of the memory space, since the reset vector needs to be stored there anyway (the 6502 reset vector is at FFFC and FFFD hex). In its present form, however, the monitor is 1022 decimal bytes long which is is long enough that the last bytes of it occupy those vector locations. The monitor that exists in the EPROM of the development system has code specific to this application, such as a jump to the memory test program and the memory test result printing program. If these were taken out the monitor would be small enough to fit in the top 1K of memory.

The monitor program expects serial input from a 6850 UART, and assumes that a 6840 timer module provides the timing signals to the UART. These expectations can be easily modified, however, to provide more flexibility to the user. All that is required is some slight modifications to the program, and it is documented well enough to allow even the novice to make the necessary

changes to the software.

The monitor uses the top 32 bytes of page zero of RAM for its workspace. This leaves the user the bottom 224 bytes of RAM. These locations were chosen because the user should not have to try to keep track of which locations the monitor is using and structure his test programs around those locations. Instead, the monitor should use locations that are very unlikely to be used by the development program.

The monitor also has its own stack, to avoid conflicts with the user stack. In the 6502, the stack is always on page one of memory. On power up or reset, the stack is initialized to the top of page one (location 01FF) and grows down toward 0100. The monitor creates its own stack, initializing the monitor stack pointer to location 010E. This allows the user nearly all of page one for his stack. Due to the limited nesting of subroutines performed by the monitor, this procedure is successful.

The monitor was written with the inexperienced user in mind. A Help instruction is provided, and a simple single breakpoint is provided as opposed to the more complex multiple level breakpoints found in more advanced monitors. Over one quarter of the existing monitor consists of messages to be sent to the screen. If these messages were removed there would be plenty of room to provide advanced monitor instructions. Perhaps by the time the user is experienced enough not to need the various messages he

will be experienced enough to modify the monitor to fit his needs. It was felt that keeping the total monitor size below 1K bytes was inportant enough to justify these choices.

The monitor does provides 7 different functions to the user. Each of these are listed and described in detail in Appendix B. If more information is desired than what is presented here, the actual monitor program is also included in Appendix D. It is documented thoroughly enough that between it and the manual all the information necessary is available.

6502 SINGLE STEPPER

This hardware device was designed and built primarily as an aid for debugging hardware, although it has proved to be an invaluable aid on many occasions when debugging software.

It was built to allow the user to execute assembly language programs one cycle at a time. Technically, it provides a means for single cycle execution of a program. It provides a means for examining memory directly, as opposed to loading memory values into registers and then viewing register the contents. It is extremely useful in locating chip select errors, since you can have the processor halt at a certain address with that address value latched onto the address lines and then observe the different chip selects. It is also very good at finding improperly connected and/or initialized peripherals, since it allows the user to examine their outputs directly while they are selected. It is usually used when the system will not run the monitor program and the reason is not easily located, or when introducing a new kind of chip into an existing system. This circuit is essentially a small hardware logic analyzer. It does not perform nearly as many different functions as the commercially available logic analyzers do, but it is far more portable and using it in conjunction with the funtions provided by the monitor program have so far proved to be more than sufficient to overcome every problem encountered.

The device allows the user to execute assembly language code one cycle at a time, run freely until a selected pattern appears on the address lines, or a combination thereof. In general, the code is allowed to run until a certain pattern appears on the address lines, at which time the processor is halted and execution continues from there one cycle at a time.

This device works due to the way the 6502 deals with its READY line. In most other memory-mapped processors (Motorola products, for instance), the ability to halt processing exists, but the address lines either revert to tri-state devices or you are only able to suspend processing for a nominal number of clock cycles (14 on the 6809). However, the 6502 handles the halt state differently. The 6502 uses two non-overlapping system clocks, 01 and 02. The rising edge of 01 initiates a fetch from memory if the R/W line is high, and initiates a memory write if the R/W line is low. The address, data, and R/W signals are all stable by the falling edge of the 01 clock. The falling edge of the 02 signal then latches the incoming data into an internal holding register ready for transfer during the subsequent 01 clock period. If the READY line is low at the rising edge of the 01 signal and the operation specified is a write, the operation continues uninhibited. However, if the operation specified is a read, the memory fetch is inhibited and the values placed on the address and other signal lines are maintained. This continues until the READY line is high at the advent of the 01 signal. Therefore, in order to cause the 6502 to step through its program

13

one cycle at a time, a mechanism for causing a single high pulse on the READY line in sync with the 02 clock is needed. This device incorporates just such a circuit.

In run mode, when no single cycling is desired, the READY line is held high. To begin single cycle execution, a mode is entered that generates a single high pulse on the READY line during the 02 clock whenever a switch is closed. To allow the user to select the address at which to halt the processor, a set of dip switches are provided. The values on the address lines are compared to the dip switch values set by the user. Since the address lines are stable during the entire time the 02 signal is high and our goal is to have the READY line low by the advent of the 01 signal, we have the whole of the positive 02 cycle (about 500 nanoseconds) to perform our compare and produce the required signal. This time is more than sufficient since we are using all 7400 series MSI and SSI parts. When the values on the address lines match the dip switch values, the READY signal is brought low and the processor is halted. A series of 7485 magnitude comparators were used to implement this part of the circuit.

There must be a way to connect the device to the circuit under test. This requirement was met by obtaining a 40 pin clip and attaching the necessary leads to it. This clip fastens directly onto the processor and leaves the single cycling device completely external to the circuit. This makes the device easily transportable between systems. Anyone with a machine using a 6502 as the CPU can make use of this device. I have used it as a

14

tool in repairing both an Apple and a (now-defunct) Ohio Scientific Computer.

In order to prevent the device itself from causing errors by excess loading of the data, address and signal lines, all the incoming signals are buffered as they enter the device. The circuit uses the 02 clock, the address lines, the data lines, and generates the READY signal. The incoming address lines and the data lines are buffered by running them through a 74LS244 non-inverting buffer chip. The 02 signal is buffered by simply inverting it twice using 7404s. The READY signal does not have to be buffered, since it is being generated by the device. Power and ground are also obtained from the CPU. This was done to increase the portability of the circuit, but it can theoretically cause problems as well since there are 24 LEDs on the device as well as 17 ICs. This presents a significant current load, and if the power supply being used in the circuit under test is too small problems could occur. This has not presented itself as a problem yet, but the possibility is there.

To maximize the usefulness of the circuit, an LED was connected to each of the address and data lines. This allows the user to see at a glance the state of the various signal lines. Using some sort of 7-segment display was considered, but it was decided that generally the desired information is the exact state of each bit, and the user would spend a lot of time converting from hex to binary. Therefore the binary information is pro-

vided, and the user spends a lot of time converting from binary to hex.

As it turned out, two completely different systems were built to provide and demonstrate the memory test capabilities, so this circuit received extensive use. It is almost a necessity to have some sort of logic analyzing capability available when building and debugging new hardware systems, and this particular circuit has worked out extremely well in providing this capability. It has saved enough time to more than justify the time and effort taken to design and construct it.

EPROM BURNER

A development system should provide the user some method for creating non-volatile copies of programs. For microprocessor-based systems, this implies the ability to program EPROMs. Since EPROM programmers are relatively simple, only a listing of the EPROM programming software (Appendix H) and a brief description of the setup are included in this report.

The Osborne I used in this system uses three banks of memory. Bank one contains the 64K bytes of RAM used by the programmer. Bank zero contains the I/O chips and the system ROMs. Bank two is the display and special character generation hardware. In order to program EPROMs, it is necessary to control directly what is being sent to the programmer. It was decided to use the parallel port on the Osborne (a Motorola 6821) and drive it from the EPROM programming software directly without involving the operating system procedures. However, this requires the ability to switch banks, and that presented some complications. The programming software contains the assembly language sub-routine that handles driving the port as data and in the initial-ization stage writes out to memory the program that it will later call. The assembly language subroutine is written in Z80 assembly language, since the Osborne uses the Z80 as its CPU. The actual EPROM programmer is connected via a Centronics printer cable to the parallel port of the Osborne, and the values appear-

ing on the output latches of the 6821 are captured by latches on the EPROM programming circuit. The EPROM programmer consists of a set of latches that are driven directly by the 6821, which is in turn driven directly by the Z80 code subroutine called repeatedly by the Pascal mother program.

The mother Pascal program provides the interface between the user and the EPROM burning program. The EPROM programmer provides most of the usual functions found on a standard EPROM progammer: the ability to verify that an EPROM has been erased, the ability to examine the contents of the EPROM, the ability to program an EPROM and verify that the programming was successful, etc. This programmer has been used many times to program the EPROM that contains the system monitor and memory test programs and has worked very satisfactorily.

REMOTE MEMORY TEST

The following sections are a description of the second part of the project. The goal of this part of the project was defined as follows:

> Provide the ability to put a 1K x 4 static RAM memory chip into the Nuclear Reactor and subject it to radiation. While undergoing irradiation, the status of the chip should be constantly monitored. A record of the number of errors and when they occurred should be kept. The system should be portable, and should be reconfigurable. It should also be easy for an inexperienced user to operate.

Several things are necessary to meet this goal. The provisions for ease of use by an inexperienced user have already been detailed in the preceeding sections. The following sections will detail the hardware and the software necessary to properly meet the testing stipulations.

Following the details of how the goals of the remote memory test project have been met will be a section outlining the future of the project and possible modifications to the test algorithm currently employed.

REMOTE MEMORY TEST SYSTEM DESCRIPTION

The following experiment was performed at the University of Wisconsin Nuclear Reactor (UWNR) to gain information on the viability of use of the facility in studying radiation effects in electronics. The UWNR is a heterogeneous pool-type reactor fueled with TRIGA-FLIP fuel which is 70% enriched in U-235. The environment is similar to the type encountered in commercial U-235, water moderated, water cooled, thermal reactors. Further details of the reactor are available at request from the reactor personnel. Metal foil activation was used to determine the neutron fluxes while an ionization chamber was used to measure the total gamma flux. Table I contains the results of the characterization of the radiation environment found at the irradiation site located in beam port #3 as shown in figure I.

The test itself involves subjecting a 2114 1K x 4 Static RAM chip to neutron and gamma radiation and logging the number of errors and the time at which they occurred. To accomplish this a hardware circuit that is connected to the Remote Memory Under Test (RMUT) and software to properly drive the hardware is required. This has been done and the RMUT system is working at the present time.

CONTROL ROD DRIVES

REACTOR BRIDGE

ROPORTIONAL COUNTER
THERMAL COLUMN
HERMAL COLUMN DOOR

BEAM PORT
SHUTTER

BEAM PORT
#3

RMUT LOCATION

ION
CHAMBE

REACTOR CO

COOLANT HEADE
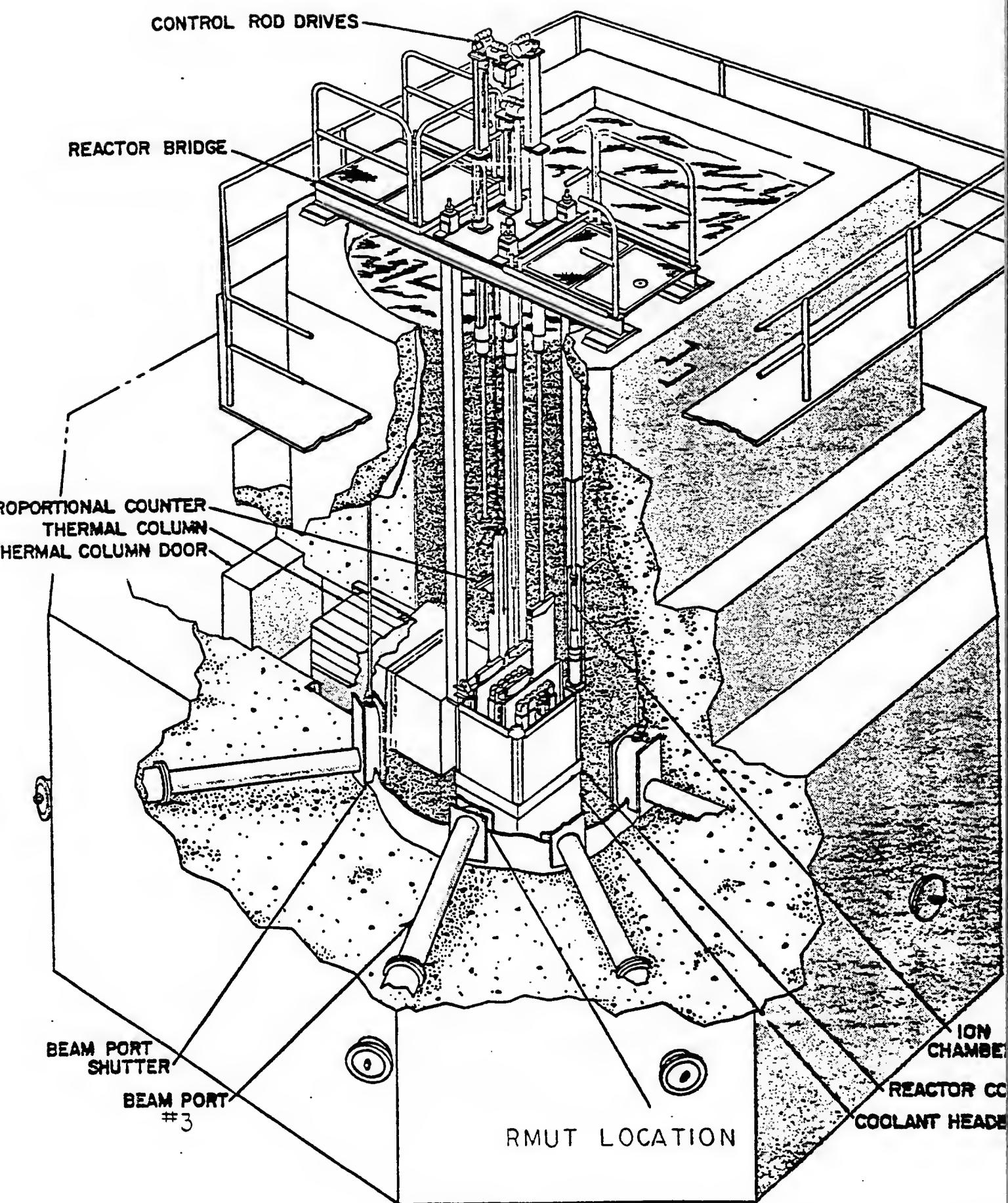
FIGURE I

```
Neutron thermal flux  = 8.33E7 +-5.8% (cm^-2 sec^-1)
                      = 0.13 Rad(Si)/hr

Neutron fast flux     = 1.38E8 +-2% (cm^-2 sec^-1)
                      = 50.0 Rad(Si)/hr

Gamma Total Dose      = 4.9 kRad(Si)/hr
                      = 4900 Rad(Si)/hr
```

Table I

```
Run# Sygnetics Motorola  Soft-errors   Occur. of 1st hard error
                                           hrs. (kRads)

  1      *                   a,b          2.43 (11.9)
  2              *            c           incomplete
  3      *                    d,e         1.14 (5.59)
  4      *                    a,b         2.68 (13.1)
  5      *                                3.0  (14.7)
  6              *            c           2.0  (9.8)
  7              *            c           incomplete
```

Comments:
    a - errors appeared as irradiation is begun.
    b - uniform occurence during irradiation.
    c - occurence coincident with hard errors.
    d - before hard errors, but after an accumulation of a dose.
    e - relatively infrequent.

Table II

Figure II

22
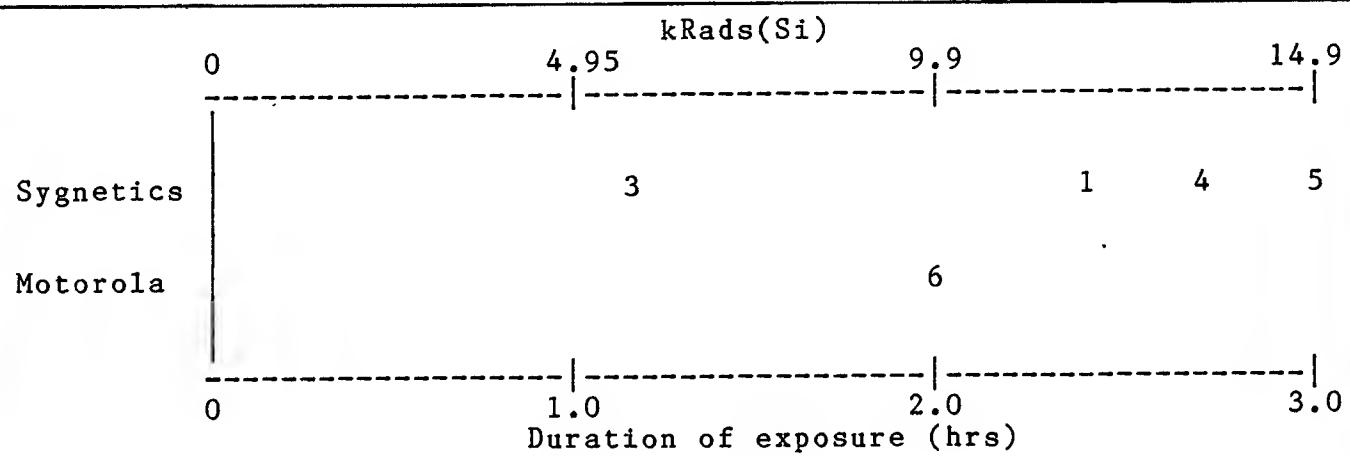
In order to determine the viability of the project an existing CPU board was extensively modified so that it could perform the bare minimum of the required tasks. Testing software was written that provided good coverage of stuck-at faults, but the coverage of the decoder and memory address register faults was difficult to obtain. Using this initial setup, the first set of results were obtained.

Regardless of the theoretical import of the results, it had been demonstrated that the RMUT concept was a viable one, and work began on a better test algorithm and a better test generation circuit, one that was designed specifically for the task at hand.

SYSTEM OPERATION

The memory chip to be irradiated is placed in neutron beam port #3 and located near the core of the reactor next to the fuel assembly, as shown in figure I. A 25 foot long cable connects the 18 pin socket in which the memory chip resides to the memory test circuit. The RMUT is controlled by a set of latches on the RMUT test circuit. These latches are connected to the address, data, and control lines of the remote memory chip. They provide an address to the remote chip and then read from or write to it depending on the function specified by the software.

Only the address, data, and R/W line values are actually modified. The function of the chip select line on the 2114 RAM is to allow the chip to be selectively enabled and disabled.

23

This capability is used when these chips are connected to a common bus, as is the case in most microprocessor applications. That kind of structure is not present here, though, since currently only one chip at a time can undergo irradiation. Therefore the chip select line can be held at a low value and does not need to change states, which is useful because reducing the number of lines changing state in the wire bundle connected to the RMUT will help reduce cross-talk noise on the signal lines. The chip select driven by a latch output and not hard-wired to ground, however, because in the future more than one chip may be placed in the reactor and tested simultaneously in which case the ability to select each one individually will be required.

The software keeps track of each error and when it occurs and stores this information in an internal buffer for later examination. The length of this buffer varies depending on how much memory is available on the system and the number of bytes necessary to completely record and define each error. (Currently 4 bytes are used: two to represent the time the error was detected and two to represent the number of errors detected during that pass.)

An error is defined as a value read from the RMUT that that deviates from the expected value. Two types of errors are possible: soft errors and hard errors. A soft error is defined as an erroneous value that is correctable, while a hard error is an erroneous value that retains its incorrect value. In the

24

first memory test program each time an incorrect value was received from the RMUT, an explicit attempt was made to correct that value. In the second (current) program, no explicit attempt at error correction is made due to the memory test algorithm used. (In this algorithm, each RMUT location gets written to and read from the several times in each pass of the program, so a soft error is discernable from a hard error by the number of times it occurs.)

Since the RMUT test circuit has no disk drives, tape drives or EEPROMS, another method of creating a permanant record of the error information was required. To accomplish this task a program to display the contents of the error buffer was written which sends the entire error buffer to the serial device. This serial device could be either a terminal or a printer. If a printer is used, it will provide a hard copy of the information. However, the data would then have to be reentered into a computer if any serious numerical analysis of the results is desired. Here is where the portability feature of the Osborne I is taken advantage of. The Osborne is simply carried in, set beside the test circuit, and connected through its serial port to the serial port of the test circuit. Using a program on the Osborne I that takes all input from its serial port and stores that information on disk allows the user to create a file on one of the disks containing all the information contained in the error buffer. Once on disk, the information is available for any kind of analysis desired.

There is no provision for keeping track of the actual time of day on the RMUT circuit. However, the software is designed to request the starting time of the test run and only begins a pass through the RMUT every 5 seconds. Thus, all the information necessary to calculate the exact time of an RMUT error is to know the number of the pass through the memory on which the error occurred and the starting time of the test. This information is all that is stored in the error buffer. Originally the test program stored the address in the RMUT of the error, the incorrect value returned by the RMUT, and what the value should have been. However, it was decided after a few test runs to concentrate on the hard errors and pay less attention to the soft errors, rendering much of this information unnecessary. It also took several bytes per error to store all of this information, and the more bytes used to quantify each error the smaller the number of errors that can be stored in the error buffer. Currently, in an effort to compress data, only the pass number and the number of errors detected during that pass are stored in the buffer.

RESULTS

A standard 1K x 4 NMOS Static RAM was chosen for the study due to its widespread usage and availability. The effects of the neutron irradiation were determined to be insignificant relative to the gamma flux. Simply stated, the effects of displacement radiation such as neutrons has greater effect on bulk properties than surface effects, whereas ionizing radiation (such as gammas)

26

has a predominate effect on the Silicon-Silicon dioxide interface which is applicable to surface phenomena devices such as NMOS transistors.

Memory chips from two vendors, Motorola and Sygnetics, were used to get the results shown in Figure II. None of these IC's were of a hardened NMOS technology and all devices failed for doses less than 100 KRads(Si); whereas, most hardened IC's will survive 1 MRad(Si) or more. However, it is noted that there are significant variations in the behavior of the IC's between vendors. Details in Figure II.

MEMORY TESTING PROGRAM

The memory testing program is the RMUT software running on the RMUT test circuit that causes the hardware to perform the test. Several different test algorithms were considered, and two different ones were implemented. These are described here.

OPERATION

The RMUT must be tested for as many errors as possible, since the exact behavior under irradiation is what we are trying to determine. We do not know before hand exactly what errors to expect. We know which ones are most likely to occur, but a complete test must check as many different variables as possible to provide the most complete picture.

The test circuit is not currently capable of testing any of the D.C. or A.C. parametric specifications of the RMUT. The only thing we can test thoroughly is the functional specifications of the memory. Does it work as specified? If a given value is stored at a memory location, does it remain there until power is turned off, or does it change with time? Are we able to store a 1 or a 0 to every memory location? Full functional testing of a memory chip is impossible in the practical sense since the complexity of such a test would be on the order of 2^n (where n is the number of memory cells in the array).

In order to test the full functionality of the RMUT, a fault

model must be specified. There are three different fault models generally in use, listed here in the order of their complexity:

1) Stuck-at Fault model. In this model, one or more of the values stored in memory are impossible to change. The value might be fixed at either a logic 0 or a logic 1. This model is also useful for modeling faults in other parts of the memory system.

2) Coupling Faults model. In this model, a change in the value stored at one location causes another location to also change its stored value. These two cells are then said to be coupled.

3) Pattern-sensitive Fault model. This model is the most complex and difficult to test. Here a change in the value stored in a memory location occurs due to some pattern of 1's and 0's stored in adjacent memory cells. The coupling fault model is actually a subset of this model.

It would be best if one could test for all of these faults. However, the best test algorithms for detecting pattern-sensitive faults are extremely long and complicated and do not provide an increase in error coverage sufficient to justify the extra time and effort involved in implementing the algorithm. Many test algorithms that detect all coupling faults will also detect all possible stuck-at faults, though, so one of these algorithms was chosen.

The first test program used in the experimental system did

not provide sufficient error coverage. It was much faster than the algorithm now in use, but since it had a random factor built into it the exact fault coverage was impossible to determine.

The test algorithm currently being used is the Nair, Thatte and Abraham testing procedure. This procedure tests for all stuck-at faults, as well as all coupling faults. It also completely tests the memory decoder and the memory address register. It is much faster than the other test algorithms that provide the same fault coverage, and is fairly straight forward to implement.

The theory behind the testing procedure is described in Abadir and Reghbati, Computing Surveys, Volume 15, No. 3, page 189, Sept. 1983. The test itself consists of an initialization step, followed by a sequence of reads from and writes to each RMUT memory cell that thoroughly test the RMUT for any decoder, memory address register or memory array stuck-at faults. After these sequences have been completed, another sequence of reads from followed immediately by two complementary writes to the RMUT are performed. This sequence tests the RMUT thoroughly for any coupling faults. (See Figure III.)

In the proof of the correctness of this test procedure, it is assumed that the memory array to be tested consists of an n x 1 array of memory cells. This is not the situation being dealt with in this case, however. In this case, the test memory is an n/4 x 4 array of memory cells. These can be treated conceptually as individual cells as far as the testing procedure goes, but some coupling fault coverage will still be lost.

30

FIGURE Ⅲ.a.

A semiconductor RAM organization at the functional level of description.
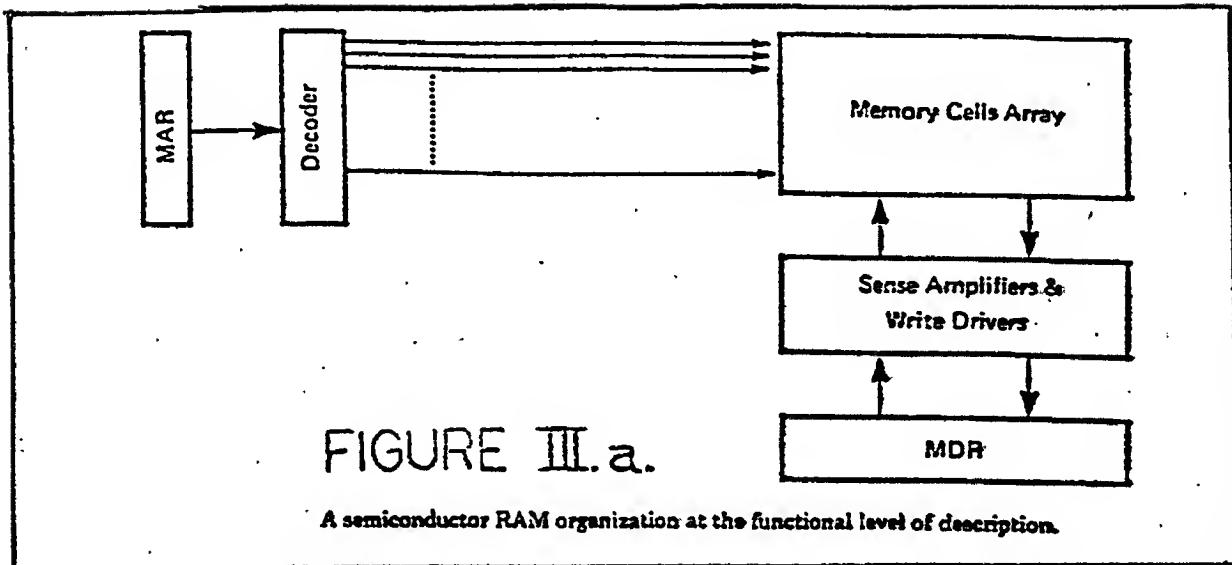
Cell Number	Initialize	Sequence 1	Sequence 2	Sequence 3	Sequence 4

→ Time

Cell Number	Sequence 5	Sequence 6	Reset	Sequence 7	Sequence 8

→ Time

Nair, Thatte, and Abraham's testing procedure [Nair et al. 1978].

R – READ
↑ – WRITE HI
↓ – WRITE LO

FIGURE Ⅲ.b

31

In the sequence that tests for coupling faults, each individual location is read, and then written both high and low. Upon performing the read of the next RMUT location, if the expected value is not received a coupling error has been detected since a write to some other RMUT address has caused the current RMUT memory location value to change. The problem faced by executing this test sequence on an n x 4 array is that when the individual cell being examined is written to, the other three cells in that row are also written to due to the inherent structure of the memory. Even though the values written back into the other three cells have not been modified, the very act of writing values back into those cells will mask any coupling that might occur between the individual cell under test and the other cells in its row.

Other possible test algorithms were examined, but none of them could surmount this problem while keeping the coupling fault coverage intact. Most of the other algorithms did no coupling fault testing at all. It was decided that even though the coupling fault coverage is reduced by using this algorithm on an n x m array, it still does provides some coupling fault coverage. It will still detect any coupling faults that might exist between the cell under test and any cell in any other row of the memory array. Therefore, it remains as the most thorough test algorithm available, which is what was defined as the objective.

This test is an O(30n) test. In our case, n is 4K, since we are conceptually treating the memory array as a 4K x 1 array. We drive the remote memory chip through a PIA, so the actual time needed to perform the entire memory test is just under 5 seconds. This is a longer than desired time for the memory test to run. It may be that in the future a test that does not provide the coupling fault detection but runs in much less time may be preferrable. Without having to do the coupling detection the test could be rewritten to run in less than 1/2 second. There are applications where this ability would be very useful.

6502 CPU BOARD

The RMUT circuit uses a MOSTEK 6502 microprocessor as its CPU. The address space is divided into 32 unique 2K x 8 sections by two 74154 4-16 line decoders. Two Motorola 6850 Universal Asynchronous Receiver/Transmitters are provided: one is used for serial terminal I/O and one is reserved for future expansion. (Attaching a printer to provide a hard copy of data has been suggested). There is a Motorola 6840 Programmable Timer Module which is used to provide the receive and transmit clocks to the 6850s. The PTM is also used to generate a 5-second signal used by the RMUT testing software as a synchronization signal and to keep track of time. Two Motorola 6821 Peripheral Interface Adapters are used to provide the actual interface with the RMUT chip. Two Hitachi 6116 2k x 8 Static RAM chips are used to provide program development workspace RAM and to provide the error buffer used by the RMUT software. One 2716 2K x 8 EPROM is provided for non-volatile program storage. Another 24-pin socket is completely wired as a 2K x 8 memory chip except for the chip select line, to allow either another ROM or another RAM to be added.

The CPU address lines are buffered by two 74LS244 non-inverting buffer chips, and the data lines are buferred by two 74LS243 bi-directional non-inverting buffer chips. This was done to make future expansion easier.

34

All output lines from the 6821s to the RMUT have 150 ohm resistors in series with the signal lines in an effort to reduce ringing. There is also an LED that is used as an indicator that the memory test program is running.

This circuit was designed in an effort to meet two design goals. One was to provide the RMUT test circuit needed by the project, and the second goal was to provide a system that could be easily expanded. There is reason to believe that more than just 1K x 4 static RAM chips will undergo irradiation in the future. Since a test facility was desired that could handle any number of different kinds of ICs, the ability to easily expand the system was very important. In an effort to meet that challenge, the memory was decoded into 32 2K blocks to allow a maximum number of different peripheral devices and memories to be added. The power supply also provides over 1 amp of current, which should be more than enough current to meet the needs of any forseeable system. It is very unlikely that more than two serial devices will be required, so two are provided. There may be a need for more PIAs, and more memory. These should have no problem fitting into the system since the address and data lines coming from the CPU are all buffered and there are more than enough chip selects to go around. All in all, it is felt that the two design goals have been satisfactorily met.

PROJECT FUTURE

There are a variety of different experiments that can be performed with this test setup. Some examples are listed here.

1). Different memory tests.

This one is very likely to occur, since the test used currently does not test for everything that might be of interest. Some examples:

   a)   Where did the error occur?
        This information might provide some insight into which part of the chip is inherently the weakest.

   b)   How many soft errors occurred?
        Current information gained from our tests indicate that certain manufacturer's chips are more prone to soft errors than others. For example, we got a very low number of soft errors when testing Motorola chips, and a much higher number when testing Sygnetics chips. To properly test the soft error condition, should more time be allowed between memory reads and writes? Does constantly updating the memory affect the soft error rate? If so, how? How much time should be allowed between passes? Determining the answer to these questions will require an entirely different memory test to be written, since the current one makes many passes very close in time to one another.

   c)   Does the RMUT recover over time or are the errors permanent?
        We have collected some date to indicate that some of the errors fix themselves after a time. Most do not, however. The program could be modified to test this phenomena.

2) Different memory chip tests

Do other types of memory chips behave the same way? The test circuit and program could be modified to test almost any kind of memory chip, although testing dynamic ram chips would be exceedingly difficult. The 6116 2K x 8 static RAM chip would be an interesting candidate. Since it is a CMOS chip, different behavior might be expected.

3) Tests of different IC's

The test circuit could be modified to test virtually any IC. It does not need to be restricted to testing memory chips. Testing the behavior of standard 7400 series TTL or CMOS chips could be accomplished with a minimum of effort. Some linear chips could also be tested, by putting an A/D converter on the test board and monitoring the output of the test chip over time. Exercising this option is under consideration by a Nuclear Engineering Ph.D. candidate.

4) Testing more than one chip at a time.

Currently, provisions are made for testing only one chip at at time. However, since the test circuit provides complete control of all of the lines of the test chip, it would take only minor modifications to the test circuit to allow several chips to be placed in the neutron flux simultaneously, and be tested in some sort of round robin manner. In order to perform this test properly, a much faster algorithm would probably need to be used, and the coupling fault coverage would have to be surrendered.

37

There are many other possible uses for this circuitry and system. Several memory manufacturing companies have been contacted and have expressed interest in supplying chips to be tested in exchange for the results of the study. Since the reactor only runs twice a week, the modification of the test system to allow more than one chip to be tested at a time is almost a necessity.

Testing several chips at once would allow several of the above tests to be performed simultaneously. Given that four chips were being irradiated, there could actually be four different types of tests running, one for each chip. Or the same test could be run on each chip, with the frequency of testing varied for each chip.

This equipment is available for use by anyone qualified to use it. It is hoped that the usage will not end with the completion of the tests currently underway.

APPENDICES

6502 Assembler Usage Information

It is assumed that the person reading this document and using this assembler is familiar with the 6502 assembly language and the numerous addressing modes available therein.

This assembler is a standard assembly language assembler. This assembler supports 8 different pseudo-operation codes and requires the assembly language instructions to conform to a restricted free format. The following is a listing of each special feature of the assembler and how to make proper use of it.

Number Representation

A number is assumed to be decimal unless it is followed by an H, which indicates the number is in hexadecimal format. For example, 20 is the number 20 decimal, while 20H is 32 decimal. When using hex numbers, if the first character of the number is in the range A-Z, it must be preceeded by a 0 so that the assembler can differentiate between hex numbers and labels. If you wish to represent the number 65,536 decimal in hex you would use 0FFFFH.

Comments

Comments may appear anywhere in a line, but they must be preceeded by a semi-colon. If a semi-colon appears in a line, everything from that point to the end of the line is ignored. Therefore, all comments must appear at the end of the line or on a line without code.

Format

This is a free-format assembler except for the following:

1) The first character in the line is used to determine the presence of a label. If there is a label it must start in column one, and if there is not a label column one must be blank.

2) Spaces are used as delimiters, so there must be at least one space between the opcode and the operand and one space between the end of the operand and the beginning of any comments.

3) Currently all labels, opcodes and operands must be in uppercase or an error will occur.

A 1

Labels

Labels, if they exist, must begin in the first column. They cannot be more than 8 characters long. They must begin with a valid letter between A and Z. Labels can be used with any instruction except the ORG pseudo-op.

Opcodes

Opcodes must be valid 6502 instructions or one of the pseudo-opcodes supported by this assembler. They cannot start in column one, although they can appear anywhere else in the line. Some must be followed by operands and others do not need to be. There are restrictions on which addressing modes are allowed with each instruction. The assembler is designed to prevent the improper assignment of an addressing mode to a given instruction. Much time can be saved by paying sufficient attention to the capabilities of each instruction before beginning, however.

Operands

Operands are required by most of the instructions in the instruction set. Some operands need to be modified by the character strings listed below, depending on the addressing mode. Simple addition and subtraction is allowed within operands, but not multiplication or division. Operands will consist of a number (decimal or hex), a label, or some combination thereof.

Addressing Modes

There are 9 different addressing modes available on the 6502. Each must have a unique representation for the assembler to be able to differentiate between them. The following is a list of the addressing modes and how each one is to be represented.

1)   Immediate addressing

The value following the instruction must be preceeded by a #. It can either be a number or a valid label.

    LDA #0A0H   ; Load the A register with 160 decimal

2) Absolute addressing

The value following the instruction is used as the 2nd and 3rd bytes of the instruction.

```
LDA EXAMPLE      ; Load A reg from the location specified by
                 ; by the label EXAMPLE
```

3) Zero Page addressing

The value following the instruction is used as the second byte of the instruction. It is assumed to lie on page zero, so the third byte is by default zero. This is specified by following the operand with the character string ,0 to signify the zero page mode.

```
LDA EXAMPLE,0    ; Load A reg from the zero page location
                 ; specified by the label EXAMPLE
```

4) Accumulator

This mode needs no operand and no operand should be given.

```
LSR              ; Shift the value in the A reg to the right
```

5) Zero Page X, Zero Page Y (Zero Page indexed)

This mode uses the zero page concept and adds either the X or the Y register value to the result depending on which is specified. It is represented by following the operand with either the string ,X,0 or ,Y,0 depending on which mode is desired.

```
LDA ZERO,X,0     ; This is zero page indexed by X
LDA ZERO,Y,0     ; This is zero page indexed by Y
```

6) Absolute X, Absolute Y (Absolute indexed)

The absolute value in the operand is indexed by either X or Y depending on which mode is specified. This is represented by following the operand with either ,X or ,Y depending on the mode desired.

```
LDA EXAMPLE,X  ; Load A with absolute address specified by
               ; label EXAMPLE indexed by X.
LDA EXAMPLE,Y  ; Same as above with Y the index instead of X
```

7) Indexed Indirect

This is a complicated addressing mode whose explanation is best left to textbooks or manuals. To represent this addressing mode the operand should be preceeded by a left parenthesis and followed by the string ,X).

```
LDA (ACIA,X)   ; This is indexed indirect representation
```

8) Indirect Indexed

This is another complicated addressing mode, whose explanation will also be left to the experts. Representing this addressing mode is accomplished by enclosing the operand in parenthesis and following with a ,Y.

```
LDA (ACIA),Y   ; This is indirect indexed representation
```

9) Indirect

This addressing mode allows for indirect jumps. It is represented by encasing the operand in parenthesis.

```
JMP (ACIA)     ; Do an indirect jump
```

Pseudo-operations

Pseudo-ops are assembly language instructions that produce no actual 6502 machine code. These operations are instructions to the assembler to perform some necessary function, such as reserving memory locations or changing the value of the PC. This assembler supports the following eight pseudo-ops.

ORG :   This sets the program counter to the value specified by the number or the label following it. There can be no label associated with this pseudo-op.

        ORG 0200H   ; This sets the PC to 200 hex

EQU :   This equates the label associated with the pseudo-op and the label or value following the pseudo-op.

        PROOF   EQU 0200H ; PROOF is now equal to 200 hex

ASC :   This takes all the information enclosed in quotation marks and converts each character into the associated ascii character representation. This is very useful for creating messages to be output to the I/O device.

        ASC "This is a demonstration"

BYT :   This will store the value(s) following the BYT pseudo-op in memory beginning at the value of the PC when the pseudo-op was encountered.

        BYT 0AH,0DH   ; Stores 10 and 13 in consecutive
                           ; bytes beginning at value of PC

RST :   This pseudo-op is used to cause the 6502 reset vector
            to contain the value specified in the instruction.
            This information appears in the .REL file.

                    RST   BEGIN   ; Set the reset vector (OFFFCH
                                         ; and OFFFDH) to value of BEGIN

IRQ :   This pseudo-op performs the same function as the RST
            pseudo-op does. However, it sets the IRQ (maskable)
            interrupt vector instead of the Reset vector.

NMI :   This pseudo-op performs the same function as the RST
            pseudo-op does. However, it sets the NMI (non-
            maskable) interrupt vector instead of the Reset vector.

END :   This pseudo-op is not really necessary, and if it is
            omitted no major problems will occur. This exists to
            tell the assembler to stop assembling code. It can be
            useful if you are debugging an isolated section of code
            and do not wish to wait for the entire file to
            assemble.

     This is all the information needed to use the 6502 assembler
correctly. In order to demonstrate some of the ideas presented
above, a sample program has been written and assembled that
contains all the pseudo-ops and all the addressing modes
discussed above. It is included so that if there are any
questions about the meanings of any of the descriptions listed
above the user will have a reference to look at. The program
does not do anything useful, so do not try running it on an
existing machine.

```
    ;

    ;       Example program to demonstrate the format and
    ;       workings of the 6502 assembler.  The first part
    ;       is correct and the last part contains errors
    ;       as an illustration of what it will and will not
    ;       accept as input.


    ;       Pseudo Opcodes

000      TEST    EQU     0C000H          ; TEST=20 Decimal
0000     TEST2   EQU     0C001H          ; TEST=20 Hex

000              ORG     0E000H          ; Initialize PC

E000     NINE1   ASC     "THIS IS A TEST" ; ASC Pseudo-op
4 48 49 53 20 49 53 20 41 20 54 45 53 54

E00E             BYT     0AH,0DH,0,1,9   ; BYT Pseudo-op
0A 0D 00 01 09

E100             RST     0E100H          ; Demonstration of the three
E033             IRQ     OPRND           ; Interrupt handling Pseudo-ops
C001             NMI     TEST2

    ;       Addressing formats

E013 0D 33 E0  NINE2   ORA     OPRND           ; Absolute
E016 0D 35 E0  NINE3   ORA     OPRND+2         ; Absolute with math

E019 5E 33 E0  NINE    LSR     OPRND,X         ; Absolute indexed by X
E01C 5E 15 E0  NINE4   LSR     TWO-10H,X       ; Absolute indexed by X (math)

E01F F9 00 C0  NINE6   SBC     TEST,Y          ; Absolute Indexed by Y
E022 F9 20 C0  NINE7   SBC     TEST+32,Y       ; Absolute Indexed by Y (math)

E025 A5 37     TWO     LDA     LABEL,0         ; Zero page
E027 15 37             ORA     LABEL,X,0       ; Zero page indexed by X
E029 B6 37             LDX     LABEL,Y,0       ; Zero page indexed by Y

E02B 6C 33 E0  THREE   JMP     (OPRND)         ; Indirect
E02E 6C 1D E0          JMP     (OPRND-16H)     ; Indirect with math

E031 01 33             ORA     (OPRND,X)       ; indexed indirect
E033 11 33     OPRND   ORA     (OPRND),Y       ; indirect indexed

E035 A2 FF             LDX     #0FFH           ; Immediate

E037 0A        LABEL   ASL                     ; implied

E038 00                BRK                     ; accumulator
```

```
                      ;              Miscellaneous examples

039 A2 37                    LDX        #LABEL           ; Load X with address of LABEL
03B 30 F3                    BMI        OPRND-3          ; Branch with math
03D FO FD                    BEQ        -3               ; Branch backward without label
03F DO OA                    BNE        10               ; Branch forward without label

041..AD A4 06                LDA        1700             ; Load from 1700 decimal
044 AD 23 23                 LDA        02323H           ; Load from 2323 hex

                      ;              Examples of the nine different kinds of errors
```

omment without remembering to preceed with semi-colon
** invalid opcode ***

omment without remembering to preceed with semi-colon
** invalid opcode ***

```
047 **            |  Comment without remembering to preceed with semi-colon
```

```
        STX       NOTHING,0
** operand not in table ***
```

```
04A 86            |           STX        NOTHING,0
```

```
        STX       LABEL+2k,0
*** illegal hex value in operand ***
```

```
04C 86 37         |           STX        LABEL+2k,0
```

```
        ORA       LABEL,Y,0
** invalid addressing mode ***
```

```
                  |
```

```
        EQU       20
*** must have lable with EQU ***
```

```
                  |
```

```
        BPL       TST
** branch out of bounds ***
```

```
04E 10 37         |           BPL        TST
```

```
000               |           ORG        0
```

A 8

NE7   AND   #1
* multiply defined lable ***

          |

T   ASC   HELLO THERE"
* invalid structure in ASC ***

00 | TST   ASC   HELLO THERE"
                |

mbol table

TEST	C000	TEST2	C001	NINE1	E000	NINE2	E013
NINE3	E016	NINE	E019	NINE4	E01C	NINE6	E01F
NINE7	E022	TWO	E025	THREE	E02B	OPRND	E033
LABEL	E037	Comment	E047	NINE7	0000	TST	0002

tal number of errors : 9

A 9

This document provides a detailed description of each of the seven functions available on the monitor.

L   — Download From External Source

This routine allows the user to download code from an external device directly into the memory of the development system. The format used is not Intel Hex format. This modification probably should be made. Currently, however, it expects incoming data to meet the following specifications:

All characters preceeding the first "address coming" flag (the >) are ignored. Once the "address coming" flag has been received, the next four valid ASCII characters are converted into an address at which to begin storing data. Characters are then read from the external device, converted to a data byte, and stored sequentially beginning at the location specified and continuing until the "end of transmission" flag (the <) or another "address coming" flag is received. Any other non-valid hex character is ignored. Upon receiving the "end of transmission" character control is returned to the monitor, while the "address coming" character causes the whole cycle to begin again.

The incoming data is expected in pairs of characters. Originally error-checking was included, but in the interest of saving code space it was eliminated. Due to the low transmission rate (1200 baud), so far there have been no transmission errors. If there is a need to go to a higher baud rate in the future, the error-checking may need to be reincorporated.

Example :

```
> 0200                  ; start loading
AD 00 E0 8D 01 E0       ; load from memory and write to memory
<                       ; all done
```

This will cause the machine code for a load and a store to be loaded into memory beginning at location 0200 hex. The comments will all be ignored, since they are not valid ASCII characters.

Mxxxx – Examine memory beginning at location xxxx.

This routine allows the user to examine the current contents of memory beginning at the location specified following the M command. The user can also modify what is stored at that location if the memory locations specified are alterable. The user advances to the next memory location by pressing the space bar. To change the value stored in a location, the user types in the hex values of the data to be inserted following the displayed contents of that location. If the user makes a mistake in entering data or wishes to exit this routine, pressing the carriage return key will cause a return to the main part of the monitor.

Example : (User input represented by underlined chars)

[M0200 AD A9 00 <CR>
[M0200 A9

In this example, the contents of memory location 0200 are examined, returning the hex value AD. In order to change the contents of that location to an A9 hex, the A9 is entered after the display of the AD. To verify that the change had been made, the carriage return key is hit and the M routine is re-entered. The contents of location 0200 can now be seen to have been changed to A9 hex, as desired.

Gxxxx – Begin execution at location xxxx

This routine allows the user to execute code stored in the memory of the system. Usually, the code will have been down-loaded from an external source, although for short tests the code can be entered into memory by hand using the M command. This routine returns the user stack pointer to the top of page one of RAM, and transfers program control to the location specified by the xxxx value.

Example

G0200   ; This begins executing the code stored at 0200

Bxxxx – Insert a breakpoint at location xxxx

This routine allows the user to insert a software interrupt into a program stored in the system memory. The software interrupt opcode is inserted at location xxxx and the value of the byte at that location is saved for future replacement.

When in the execution of the code the Break opcode is encountered, it will cause a software interrupt which sends the program to a routine that saves the current state of the machine:

The Stack Pointer, the A, X and Y registers, the Status Register and the Program counter values are all stored in reserved main memory locations set aside by the monitor. The software interrupt opcode is replaced by the value it originally contained. After all of this has been done, control is returned to the monitor. This allows the user to examine the complete state of the machine at the time of the software interrupt. This ability is extremely useful in debugging complex software.

Example

B0200   ; Insert a breakpoint at location 0200

C   - Continue from last breakpoint

This routine allows the user to continue the execution of his program from the spot at which the software interrupt was encountered. The users environment at the time of the interrupt is reconstructed from the data stored in the reserved memory locations and program control is given back to the interrupted process. This procedure allows the user to not only view the state of the machine when the interrupt occurred, but modify it as well. By changing the values of the registers and Stack Pointer using the R command, the user is actually modifying the environment that is to be reconstructed for his process. This allows the user great flexibility in debugging his program, since he can create different environments and observe how the program responds.

·Example

C   ; This will continue execution of the program

R   - Display the contents of the registers

This routine allows the user to examine and modify the contents of the registers as they appeared at the time of the last software interrupt. It is functionally identical to the M command, with the procedural steps needed to observe and modify register contents being the same. The only difference is that the R command examines and modifies the known reserved memory locations containing the saved register values and tells you as you step through them which register value you are looking at, while the M command allows you to examine and modify any location in memory. Using the M command and knowing the location of the reserved memory locations allows the user to accomplish the same thing as using the R command, but requiring him to do it that way would place an unnecessary burden on the user. Instead, this much simpler means is provided for observing and modifying the state of the machine.

Example

[<u>R</u> X= 00 <u>FF</u> Y= 00 <u><CR></u>
[<u>R</u> X= FF

This example demonstrates how to change the value in the X register. As stated above, the format is the same as that for the M command.

H   - Display Help Message

This routine is provided for the user who wishes a list of the commands available to him. Since this system is designed for the inexperienced user, it was decided to leave this function in and cut down on the more complex features of a monitor. A monitor is of no use if the person using it cannot make it work.

Example

[<u>H</u>
           6502 DEVELOPMENT SYSTEM

L   - Download from external source
Mxxxx   - Examine memory location xxxx
Gxxxx   - Begin execution at location xxxx
Bxxxx   - Breakpoint at location xxxx
C   - Continue from last breakpoint
R   - Display register contents
H   - This message

Added features

T   - Execute remote memory test
P   - Print out results of test

The last part of this message concerns the actual current implementation of the monitor in the development system. If the monitor was to be used on a different system, these added features would not be included.

```pascal
program assembler;
```

```pascal
type char_set = set of char;
     input_string = string[80];
     word2    = string[2];
     word3    = string[3];
     word4    = string[4];
     word8    = string[8];
     word14   = string[14];
     word12   = string[12];
     word70   = string[70];
     word210  = string[210];
```

```
{ These are all the available opcode values }

     all_opcodes = (clc,cld,cli,clv,dex,dey,inx,iny,nop,pha,php,pla,plp,
     rti,rts,sec,sed,sei,tax,tay,tsx,txa,txs,tya,brk,bcc,bcs,beq,
     bmi,bne,bpl,bvc,bvs,adc,unt,asl,bit,cmp,cpx,cpy,dec,eor,inc,
     jmp,jsr,lda,ldx,ldy,lsr,ora,rol,ror,sbc,sta,stx,sty,asc,byt,equ,org,
     nmi,irq,ent,rst,bad);

{ These are all the different available addressing modes }

     all_modes = (abz,zrp,zpx,zpy,abx,aby,ind,iix,iiy,acc,imm,rel,imp,xxx,psd,
                                                        bte,ask,orq);

{ These are the qualifiers that allow ease of identifying the modes }

     qual = (zerp,zerx,zery,indr,indx,indy,absl,absx,absy,immd,done);

{ This is a record created by the first pass for each lable.  This collection
  of records constitutes the symbol table.  Being dynamic in nature, there is
  no waste of space, and no limit to the number of labels a program can use. }

     lable_pointer = ^lable_record;
     lable_record = record
                       lable : word8;
                       lable_value : real;
                       next : lable_pointer;
                       end;

{ This is a record also created by the first pass.  This contains error
  information about the first line.  The type of error detected during
  the first pass is stored in this record.  One of these is created for each
  line in the assembly language program.                                      }

     status_pointer = ^status_record;
     status_record = record
                       kind : integer;
                       next : status_pointer;
                       end;
```

{ These are the variables used by this program.   These are all global
  variables, most of which are self-explanatory.   }

```
var   f1,f2,f3                                : text;
      filename                               : word14;
      in_line,in_line1                       : input_string;
      lable_chars,valid_chars,skip_chars     : char_set;
      end_chars,operand_chars,valid_bytes    : char_set;
      lable,lead_in                          : word8;
      operand                                : word70;
      real_operand                           : word12;
      op_code,operand1,operand2              : word2;
      result                                 : word4;
      is_label,found_label,next_valid        : boolean;
      valid_line,first_pass,quit,ok          : boolean;
      current_lable,lable_head,found_element : lable_pointer;
      current_status,status_head             : status_pointer;
      table                                  : array [all_opcodes] of word3;
      error_list                             : array [1..10] of word70;
      opcode                                 : all_opcodes;
      multable                               : array [clc..sty,abz..imp] of word2;
      ends                                   : array [qual] of word4;
      mode                                   : all_modes;
      threes                                 : set of all_modes;
      accum,singles,branches                 : set of all_opcodes;
      pc,symbol_value,temp                   : real;
      length_line,tot,byte_no                : integer;
      errors,i,j,k,error_type                : integer;
      long_string                            : word210;
```

C 3

```
{*********************************************************************

                      Beginning of Procedures

 *********************************************************************

         This procedure initializes the tables and error arrays used by the
 assembler.  This is needed since pascal has no kind of Data statement like
 some other languages do.  This two dimensional table makes a nice data
 structure for an assembler, since you can calculate the opcode and operand
 and tell immediately if that entry is a valid one.                       }

procedure initialize;

var i : all_opcodes;
    j : all_modes;

begin

{ Convert incoming ascii character string to set value }

   table[clc]:='CLC';  table[cld]:='CLD';  table[cli]:='CLI';  table[clv]:='CLV';
   table[dex]:='DEX';  table[dey]:='DEY';  table[inx]:='INX';  table[iny]:='INY';
   table[nop]:='NOP';  table[pha]:='PHA';  table[php]:='PHP';  table[pla]:='PLA';
   table[plp]:='PLP';  table[rti]:='RTI';  table[rts]:='RTS';  table[sec]:='SEC';
   table[sed]:='SED';  table[sei]:='SEI';  table[tax]:='TAX';  table[tay]:='TAY';
   table[tsx]:='TSX';  table[txa]:='TXA';  table[txs]:='TXS';  table[tya]:='TYA';
   table[brk]:='BRK';

   table[bcc]:='BCC';  table[bcs]:='BCS';  table[beq]:='BEQ';  table[bmi]:='BMI';
   table[bne]:='BNE';  table[bpl]:='BPL';  table[bvc]:='BVC';  table[bvs]:='BVS';

   table[adc]:='ADC';  table[unt]:='AND';  table[asl]:='ASL';  table[bit]:='BIT';
   table[cmp]:='CMP';  table[cpx]:='CPX';  table[cpy]:='CPY';  table[dec]:='DEC';
   table[eor]:='EOR';  table[inc]:='INC';  table[jmp]:='JMP';  table[jsr]:='JSR';
   table[lda]:='LDA';  table[ldx]:='LDX';  table[ldy]:='LDY';  table[lsr]:='LSR';
   table[ora]:='ORA';  table[rol]:='ROL';  table[ror]:='ROR';  table[sbc]:='SBC';
   table[sta]:='STA';  table[stx]:='STX';  table[sty]:='STY';

   table[asc]:='ASC';  table[byt]:='BYT';  table[equ]:='EQU';  table[org]:='ORG';
   table[nmi]:='NMI';  table[irq]:='IRQ';  table[rst]:='RST';  table[ent]:='END';
   table[bad]:='   ';
```

```pascal
{ Initially, set entire array to "invalid opcode" }

    for i:=clc to sty do
      for j:=abz to xxx do multable[i,j]:='xx';

    multable[clc,imp]:='18';  multable[cld,imp]:='D8';  multable[cli,imp]:='58';
    multable[clv,imp]:='B8';  multable[dex,imp]:='CA';  multable[dey,imp]:='88';
    multable[inx,imp]:='E8';  multable[iny,imp]:='C8';  multable[nop,imp]:='EA';
    multable[pha,imp]:='48';  multable[php,imp]:='08';  multable[pla,imp]:='68';
    multable[plp,imp]:='28';  multable[rti,imp]:='40';  multable[rts,imp]:='60';
    multable[sec,imp]:='38';  multable[sed,imp]:='F8';  multable[sei,imp]:='78';
    multable[tax,imp]:='AA';  multable[tay,imp]:='A8';  multable[tsx,imp]:='BA';
    multable[txa,imp]:='8A';  multable[txs,imp]:='9A';  multable[tya,imp]:='98';
    multable[brk,imp]:='00';


    multable[bcc,rel]:='90';  multable[bcs,rel]:='80';  multable[beq,rel]:='F0';
    multable[bmi,rel]:='30';  multable[bne,rel]:='D0';  multable[bpl,rel]:='10';
    multable[bvc,rel]:='50';  multable[bvs,rel]:='70';


    multable[adc,abz]:='6D';   multable[unt,abz]:='2D';   multable[asl,abz]:='0E';
    multable[bit,abz]:='2C';   multable[cmp,abz]:='CD';   multable[cpx,abz]:='EC';
    multable[cpy,abz]:='CC';   multable[dec,abz]:='CE';   multable[eor,abz]:='4D';
    multable[inc,abz]:='EE';   multable[jmp,abz]:='4C';   multable[jsr,abz]:='20';
    multable[lda,abz]:='AD';   multable[ldx,abz]:='AE';   multable[ldy,abz]:='AC';
    multable[lsr,abz]:='4E';   multable[ora,abz]:='0D';   multable[rol,abz]:='2E';
    multable[ror,abz]:='6E';   multable[sbc,abz]:='ED';   multable[sta,abz]:='8D';
    multable[stx,abz]:='8E';   multable[sty,abz]:='8C';


    multable[adc,zrp]:='65';   multable[unt,zrp]:='25';   multable[asl,zrp]:='06';
    multable[bit,zrp]:='24';   multable[cmp,zrp]:='C5';   multable[cpx,zrp]:='E4';
    multable[cpy,zrp]:='C4';   multable[dec,zrp]:='C6';   multable[eor,zrp]:='45';
    multable[inc,zrp]:='E6';   multable[lda,zrp]:='A5';   multable[ldx,zrp]:='A6';
    multable[ldy,zrp]:='A4';   multable[lsr,zrp]:='46';   multable[ora,zrp]:='05';
    multable[rol,zrp]:='26';   multable[ror,zrp]:='66';   multable[sbc,zrp]:='E5';
    multable[sta,zrp]:='85';   multable[stx,zrp]:='86';   multable[sty,zrp]:='84';


    multable[adc,zpx]:='75';   multable[unt,zpx]:='35';   multable[asl,zpx]:='16';
    multable[cmp,zpx]:='D5';   multable[dec,zpx]:='D6';   multable[eor,zpx]:='55';
    multable[inc,zpx]:='F6';   multable[lda,zpx]:='B5';   multable[ldy,zpx]:='B4';
    multable[lsr,zpx]:='56';   multable[ora,zpx]:='15';   multable[rol,zpx]:='36';
    multable[ror,zpx]:='76';   multable[sbc,zpx]:='F5';   multable[sta,zpx]:='95';
    multable[sty,zpx]:='94';


    multable[ldx,zpy]:='B6';   multable[stx,zpy]:='96';

    multable[adc,abx]:='7D';   multable[unt,abx]:='3D';   multable[asl,abx]:='1E';
    multable[cmp,abx]:='DD';   multable[dec,abx]:='DE';   multable[eor,abx]:='5D';
    multable[inc,abx]:='FE';   multable[lda,abx]:='BD';   multable[ldy,abx]:='BC';
    multable[lsr,abx]:='5E';   multable[ora,abx]:='1D';   multable[rol,abx]:='3E';
    multable[ror,abx]:='7E';   multable[sbc,abx]:='FD';   multable[sta,abx]:='9D';


    multable[adc,aby]:='79';   multable[unt,aby]:='39';   multable[cmp,aby]:='D9';
    multable[eor,aby]:='59';   multable[lda,aby]:='B9';   multable[ldx,aby]:='BE';
    multable[ora,aby]:='19';   multable[sbc,aby]:='F9';   multable[sta,aby]:='99';
```

C 5

```
  multable[jmp,ind]:='6C';

  multable[adc,iix]:='61';    multable[unt,iix]:='21';    multable[cmp,iix]:='C1';
  multable[eor,iix]:='41';    multable[lda,iix]:='A1';    multable[ora,iix]:='01';
  multable[sbc,iix]:='E1';    multable[sta,iix]:='81';

  multable[adc,iiy]:='71';    multable[unt,iiy]:='31';    multable[cmp,iiy]:='D1';
  multable[eor,iiy]:='51';    multable[lda,iiy]:='B1';    multable[ora,iiy]:='11';
  multable[sbc,iiy]:='F1';    multable[sta,iiy]:='91';

  multable[asl,acc]:='0A';    multable[lsr,acc]:='4A';    multable[rol,acc]:='2A';
  multable[ror,acc]:='6A';

  multable[adc,imm]:='69';    multable[unt,imm]:='29';    multable[cmp,imm]:='C9';
  multable[cpx,imm]:='E0';    multable[cpy,imm]:='C0';    multable[eor,imm]:='49';
  multable[lda,imm]:='A9';    multable[ldx,imm]:='A2';    multable[ldy,imm]:='A0';
  multable[ora,imm]:='09';    multable[sbc,imm]:='E9';

{ These are the different possible operand modifiers depending on which
  addressing mode you are in.                                                    }

  ends[zerp]:=',0';   ends[zerx]:=',X,0';   ends[zery]:=',Y,0';
  ends[indr]:='()';   ends[indx]:='(,X)';   ends[indy]:='(),Y';
  ends[absl]:='';     ends[absx]:=',X';     ends[absy]:=',Y';
  ends[immd]:='#';

{ These are the 9 errors that the assembler will catch.                         }

  error_list[1]:='*** invalid opcode ***';
  error_list[2]:='*** illegal hex value in operand ***';
  error_list[3]:='*** operand not in table ***';
  error_list[4]:='*** branch out of bounds ***';
  error_list[5]:='*** multiply defined lable ***';
  error_list[6]:='*** must have lable with EQU ***';
  error_list[7]:='*** cannot use lable with ORG ***';
  error_list[8]:='*** invalid structure in ASC ***';
  error_list[9]:='*** invalid addressing mode ***';

end;
```

```
{*****************************************************************************

Advance the pointer in the.current line to the next valid character.}

procedure skip_blanks;

begin
  while (in_line[i] in skip_chars) do i:=i+1;
  if (in_line[i] in valid_chars) then next_valid:=true else next_valid:=false;
end;

{*****************************************************************************

Print out the type of error encountered, update status record if second pass }

procedure print_error;

begin
  if first_pass then current_status^.kind:=error_type
  else
  begin
    if current_status^.kind=0 then current_status^.kind:=error_type;
    writeln;
    writeln(in_linel);
    writeln(error_list[current_status^.kind]);
    writeln(f2);
    writeln(f2,in_linel);
    writeln(f2,error_list[current_status^.kind]);
    writeln(f2);
    errors:=errors+1;
  end;
end;

{*****************************************************************************

Global var LABLE gets lable in line.}

procedure get_lable;

var j,k : integer;

begin
  j:=i;
  while (in_line[i] in valid_chars) do i:=i+1;
  k:=i-j;
  lable:=copy(in_line,j,k);
end;
```

C 7

```
{****************************************************************************

Global var OPCODE gets the opcode in line.   Opcode table searched, error
flagged if invalid opcode.                                                  }

procedure get_opcode;

var temp : word3;

begin
  temp:=copy(in_line,i,3);
  i:=i+3;

  opcode:=clc;
  while (opcode<>bad) and (table[opcode]<>temp) do opcode:=succ(opcode);
  if opcode=bad then
  begin
    error_type:=1;
    print_error;
  end;
end;
```

C 8

```
{*********************************************************************

Global var OPERAND gets operand in line.  Operand means everything from the
end of the opcode to the beginning of the comment or end of line.  ASC and
BYT have to be handled somewhat differently, since they have the potential
of being much longer than the usual operand.                              }

procedure get_operand;

var j,k : integer;

begin
  j:=i;
  k:=length(in_line);
  case opcode of
    asc : begin
            if in_line[i]='"' then          { Get ascii string }
            begin
              i:=i+1;
              while (in_line[i]<>'"') and (i<k) do i:=i+1;
              operand:=copy(in_line,j+1,i-j+1);
            end
            else
            begin
              error_type:=8;             { If not pair of ", error }
              print_error;
            end;
          end;

    byt : begin
            while in_line[i] in valid_bytes do i:=i+1;
            operand:=copy(in_line,j,i-j);     { get ascii string of bytes }
          end;

  else
    while (in_line[i] in valid_chars) do i:=i+1;
    operand:=copy(in_line,j,i-j);
  end;

  operand:=operand+chr(1);
end;
```

C 9

```
{***********************************************************************

Break down the line into its constituent components.  This routine calls the
routines listed above and uses them to parse the line into label, opcode and
operand.                                                                  }

procedure parse_line;

begin
  operand:='';
  skip_blanks;

  if next_valid then
  begin
    valid_line:=true;
    if (in_line[1] in lable_chars) then
    begin                                     { if label, get it }
      is_label:=true;
      get_lable;
    end
    else is_label:=false;

    skip_blanks;

    if next_valid then get_opcode;        { if opcode, get it }

    skip_blanks;

    if next_valid then get_operand;       { if operand, get it }
  end
  else valid_line:=false;
end;
```

```
{*******************************************************************

Convert real number to 4 ascii chars.}

function conhex( value : real) : word4;

var i,temp,temp2 : integer;
    result : word4;

begin
  result:='';

{ Need to convert real number to integer number }

  if value>32767.0 then temp2:=trunc(value-65535.0)-1 else temp2:=trunc(value);

  for i:=4 downto 1 do
  begin
    temp:=temp2;
    temp:=temp and 15;
    if temp>9 then temp:=temp+7;          { convert number to hex  }
    insert(chr(temp + 48),result,1);      { convert hex to ascii   }
    temp2:=temp2 shr 4;
  end;
  conhex:=result;
end;
```

```
{*********************************************************************
Convert ascii string to real number.  Used to convert operands to numerical
values.  Since some operands come in as negative numbers, must trap for these
cases.                                                                  }

function make_number(operand : word12) : real;

var  minus,wrong : boolean;
     i,temp,mult : integer;
     count,value : real;

begin
  value:=0.0;
  count:=1.0;
  if operand[1]='-' then
  begin
    minus:=true;                          { If minus sign, log fact }
    delete(operand,1,1);
  end
  else minus:=false;

  temp:=length(operand);
  if operand[temp]='H' then
  begin
    mult:=16;                             { If hex number, log fact }
    delete(operand,temp,1);
  end
  else mult:=10;
{ This routine does the actual conversion, which differs between decimal
  and hex numbers.                                                        }

  wrong:=false;
  for i:=length(operand) downto 1 do
  begin
    temp:=ord(operand[i])-48;
    if ((mult=10) and ((temp>9) or (temp<0))) then wrong:=true;
    if temp>9 then temp:=temp-7;
    if ((temp>15) or (temp<0)) and (mult=16) then wrong:=true;
    value:=value + (count*temp);
    count:=count*mult;
  end;

  if wrong then                    { If illegal value detected in string, error }
  begin
    error_type:=2;
    print_error;
    value:=0.0;
  end
  else if minus then value:=0.0-value;  { if minus, subtract }
  make_number:=value;
end;
```

C 12

```
{*********************************************************************

Parse operand and produce Global REAL_OPERAND, calculate the addressing mode
and place in global MODE.  If there is an error, MODE will leave as xxx.      }

procedure parse_operand;

var  i,j : integer;
     qualifiers : word70;
     k : qual;

begin
  mode:=xxx;

  if length(operand)=0 then          { if no operand should be one of these modes }
  begin
    if (opcode in accum) then mode:=acc else
      if (opcode in singles) then mode:=imp;
  end
  else
  begin
    if ((operand[1]='#') or (operand[1]='(')) then i:=2 else i:=1;
    j:=i;
    while (operand[j] in operand_chars) do j:=j+1;
    if i=2 then j:=j-2 else j:=j-1;
    real_operand:=copy(operand,i,j);
    qualifiers:=operand;
    delete(qualifiers,i,j);          { extract the real operand from out of this }
    j:=length(qualifiers);           { mess and calculate which addressing mode }
    delete(qualifiers,j,1);
    k:=zerp;
    while (ends[k]<>qualifiers) and (k<>done) do k:=succ(k);
    case k of
      zerp : mode:=zrp;
      zerx : mode:=zpx;
      zery : mode:=zpy;
      absl : mode:=abz;
      absx : mode:=abx;
      absy : mode:=aby;
      indr : mode:=ind;
      indx : mode:=iix;
      indy : mode:=iiy;
      immd : mode:=imm;
    end;
  end;

  if (opcode in branches) then
    if ((mode=abz) or (mode=imm)) then mode:=rel;

end;
```

```
{***********************************************************************

Search the symbol table for the occurance of target, Global SYMBOL_VALUE gets
value of lable in table.   Globals FOUND_ELEMENT, FOUND_LABLE also affected.}

procedure search_lable(target : word12);

var temp : lable_pointer;
    temp1,temp2 : integer;
    temp3,tail : word12;
    temp4 : real;

begin
  temp1:=length(target);

{ Since math is allowed in operand, must trap for imbedded operators. }

  temp2:=pos('+',target);
  if temp2=0 then temp2:=pos('-',target);
  if temp2<>0 then
  begin
    temp3:=copy(target,1,temp2-1);
    tail:=copy(target,temp2,temp1-temp2+1);
    target:=temp3;
    if tail[1]='+' then delete(tail,1,1);
  end;

{ Once the actual label part of the opcode is extracted, look for it }

  temp:=lable_head;
  found_element:=temp;
  found_label:=false;
  while (temp^.next <> nil) and (not found_label) do
  begin
    found_element:=temp;
    if temp^.lable = target then found_label:=true;
    temp:=temp^.next;
  end;

  symbol_value:=found_element^.lable_value;
  if temp2<>0 then symbol_value:=symbol_value+make_number(tail);
end;
```

C 14

```
{*******************************************************************

Put new entry in symbol table.}

procedure fill_lable(symbol : word8; value : real);

var temp : lable_pointer;

begin
  new(temp);
  with current_lable^ do
  begin
    lable:=symbol;
    lable_value:=value;
    next:=temp;
  end;

  temp^.next:=nil;
  current_lable:=temp;
end;

{*******************************************************************

Move the current status pointer to the next record.  Initialize status to
-1 so that items after the END statement will still be written to .LST file }

procedure update_status;

var temp : status_pointer;

begin
  new(temp);
  temp^.next:=nil;
  temp^.kind:=-1;
  current_status^.next:=temp;
  current_status:=temp;
end;
```

```
(*******************************************************************

Calculate the ascii value of opcode, place it in Global OP_CODE.  Calculate
the amount to increment the pc,. place in Global BYTE_NO.}

procedure calc_opcode;

begin
  if mode=xxx then op_code:='xx'
  else
  begin
    op_code:=multable[opcode,mode];
    if op_code='xx' then
    begin
      error_type:=9;           { If illegal opcode, flag error }
      print_error;
    end;

    case mode of                { calculate PC offset }
      abz,abx,aby,ind : byte_no:=3;
      imm,zrp,iix,iiy,zpx,rel,zpy : byte_no:=2;
      acc,imp : byte_no:=1;
    end;
  end;
end;
```

```
{****************************************************************

Calculate the second and third bytes of the instruction, if they exist.}

procedure calc_operand;

var real_temp,real_temp2,value : real;
    return_value : real;
   .specials : set of all_modes;
    result : word4;

begin
  specials:=[acc,imp,psd,orq,ask,bte,xxx];
  operand1:='   ';
  operand2:='   ';
  if not (mode in specials) then
  begin
    if real_operand[1] in lable_chars then
    begin

{ if the real operand is a label, treat differently than if number }

      search_lable(real_operand);
      if not found_label then          { undefined label error }
      begin
        error_type:=3;
        print_error;
      end
      else
      begin
        if mode<>rel then result:=conhex(symbol_value)
        else
        begin                              { if branch, calculate offset }
          value:=symbol_value-pc-2.0;
          if  (value<-128.0) or (value>127.0) then
          begin
            error_type:=4;                { if offset to big, error }
            print_error;
          end
          else result:=conhex(value+32768.0);
        end;

        operand1:=copy(result,3,2);

        if (mode in threes) then operand2:=copy(result,1,2);
      end;
    end
```

C 17

```
      else            { if not label, easier to deal with }
    begin
      value:=make_number(real_operand);
      if mode<>rel then
      begin
        result:=conhex(value);
        operand1:=copy(result,3,2);
        if byte_no=3 then operand2:=copy(result,1,2);
      end
      else
      begin
        if (value<-128.0) or (value>127.0) then
        begin
          error_type:=4;        { same problem with branches }
          print_error;
        end
        else operand1:=copy(conhex(value+32768.0),3,2);
      end;
    end;
  end;
  lead_in:=op_code+' '+operand1+' '+operand2;
end;

{ ********************************************************************************

Some instructions do not have addressing modes; this function assumes none
exist in the operand and does a straight calculation of the operand.  This is
usually used by the Pseudo-opcodes.                                           }

function get_value : real;

begin
  delete(operand,length(operand),1);
  if operand[1] in lable_chars then
  begin
    search_lable(operand);
    if not found_label then            { Search for label in symbol table }
    begin
      error_type:=3;
      print_error;
    end
    else get_value:=symbol_value;
  end
  else get_value:=make_number(operand);
end;
```

C 18

```
{*********************************************************************

Print out to the files in proper format.  Global LENGTH_LINE affected.}

procedure print_spc(which : word210);

var i : integer;
    temp : word70;
    temp2 : word210;

begin
  temp2:=which;
  i:=length(which);
  while i>70 do
  begin
    temp:=copy(which,1,69);
    writeln(f2,temp);
    delete(which,1,69);
    i:=length(which);
  end;
  if i>0 then writeln(f2,which);

  i:=length(temp2);
  j:=69-length_line;
  while i>j do
  begin
    temp:=copy(temp2,1,j);
    writeln(f3,temp);
    delete(temp2,1,j);
    length_line:=0;
    i:=i-j;
    j:=69-length_line;
  end;
  write(f3,temp2);
  length_line:=length_line+length(temp2);
end;
```

```
{*****************************************************************

Send a copy of the symbol table to both the .LST file and the screen.        }

procedure output_symbol_table;

var temp : lable_pointer;
    i : integer;

begin
  temp:=lable_head;
  i:=0;
  writeln;
  writeln(f2);
  writeln('symbol table');
  writeln(f2,'symbol table');
  writeln;
  writeln(f2);
  while temp^.next<>nil do
  begin
    write(temp^.lable:8,'   ',conhex(temp^.lable_value),'     ');
    write(f2,temp^.lable:8,'   ',conhex(temp^.lable_value),'      ');
    i:=i+1;
    if (i mod 4)=0 then
    begin
      writeln;
      writeln(f2);
    end;
    temp:=temp^.next;
  end;
  writeln;
  writeln(f2);
end;

{*****************************************************************

Print to the rel file, Global LENGTH_LINE incremented.}

procedure print_rel(which : word2);

begin
  if length_line>69 then
  begin
    writeln(f3);
    length_line:=0;
  end;
  write(f3,which+' ');
  length_line:=length_line+3;
end;
```

```
{***************************************************************

Open input file, traps for illegal file names.}

procedure open_infile(var filename : word14; var f1 : text);

var ok : boolean;

begin
  repeat
    write('Enter name of file : ');
    readln(filename);
    assign(f1,filename);
{$I-} reset(f1) {$I+};
    ok:=(ioresult=0);
    if not ok then writeln('file ',filename,' not found');
  until ok;
end;

{***************************************************************

Open output files.}

procedure open_outfile(filename : word14; var f2,f3 : text);

var list_file,rel_file : word14;
    i : integer;

begin
  i:=pos('.',filename);
  if i<>0 then filename:=copy(filename,1,i-1);
  list_file:=filename+'.lst';
  rel_file:=filename+'.rel';
  assign(f2,list_file);
  assign(f3,rel_file);
  rewrite(f2);
  rewrite(f3);
end;
```

```
{****************************************************************************

                       Beginning of main program.

****************************************************************************}

{ Create all the necessary sets and records and initialize everything that
  needs initializing                                                         }

begin
  valid_chars:=['a'..'z','A'..'Z','0'..'9','#','(',')',',','+','-','"'];
  lable_chars:=valid_chars - ['0'..'9','#','(',')',',','+','-','"'];
  skip_chars:=[' ',':',chr(9)];
  operand_chars:=lable_chars+['0'..'9','+','-'];
  valid_bytes:=['A'..'F','H','0'..'9',','];
  singles:=[clc..brk];
  branches:=[bcc..bvs];
  accum:=[asl,lsr,rol,ror];
  threes:=[abz,abx,aby,ind];

  new(current_lable);
  current_lable^.next:=nil;
  lable_head:=current_lable;
  new(current_status);
  current_status^.next:=nil;
  current_status^.kind:=-1;
  status_head:=current_status;

  pc:=0;
  tot:=0;
  error_type:=0;
  quit:=false;

  initialize;          { Creates all the tables and arrays used by the program }

  open_infile(filename,f1);  { Open the input file }
  first_pass:=true;
```

C 22

```pascal
  while (not eof(f1)) and (not(quit)) do    { Do for all of file }
  begin
    readln(f1,in_line);
    in_line:=in_line+chr(1);   { append end-of-line character }

    write('.');
    tot:=tot+1;        { Give user idea of what line is being processed }
    if (tot mod 70)=0 then writeln;

    error_type:=0;
    i:=1;

{ Parse_line calls other procedures that wind up completely parsing the line.}

    parse_line;

{ Valid_line is a flag that represents the validity of the line (no errors)  }

    if valid_line then
    begin
      current_status^.kind:=0;    { If no errors then set no error flag }
      if is_label then
      begin
        search_lable(lable);       { If label then check for multiply defined }
        if found_label then
        begin
          error_type:=5;              { If multiply defined then error }
          print_error;
        end;                              {found_label}
      end;                              {is_label}
```

```
     case opcode of                    { Calculate new PC in these routines }

         equ : begin
                  if not is_label then
                . begin
                    error_type:=6;    { Must be label with EQU statement }
                    print_error;
                  end                             {not is_label}
                  else fill_lable(lable,get_value);  { Update symbol table }
               end;                              {equ}

         org : pc:=get_value;          { Set PC to new value }

         asc : begin
                  if is_label then fill_lable(lable,pc);
                  pc:=pc+(length(operand)-3); { First pass just calc PC }
               end;
{ This is messy routine due to way byt instruction specified.  Need to count
  number of bytes in instruction so PC can be properly incremented.         }

         byt : begin
                  j:=0;
                  if is_label then fill_lable(lable,pc); {Update table if lable}
                  k:=pos(chr(1),operand);
                  insert(',',operand,k);
                  while operand[1]<>chr(1) do
                  begin
                    k:=pos(',',operand);
                    delete(operand,1,k);
                    j:=j+1;
                  end;                          {operand[1]}
                  pc:=pc+j;
               end;                             {byt}

         irq,nmi,rst : ;               { These Pseudo-ops unaffected by first pass }

         ent : quit:=true;             { If END then stop processing }

        else                          { Not a Pseudo-op }
          if is_label then fill_lable(lable,pc);  { Fill table if lable }

          parse_operand;                { Get real operand out of instruction line }

          calc_opcode;                  { Calculate the number of bytes in the instr }

          pc:=pc+byte_no;               { Increment the PC by the proper value }
       end;                                    {case opcode}
    end;                                       {valid line}
    update_status;                      { Update this line's error status record }
end;                    {pass one}

writeln;
writeln('end of pass one');
```

C 24

```
    reset(f1);                                  { Reset file so we can go through it again }
    open_outfile(filename,f2,f3); { Open the two output files }
    first_pass:=false;
    current_status:=status_head;    { Point back at first record of error status }

    length_line:=1;
    errors:=0;
    tot:=0;                              { Reset relevant values }
    pc:=0;

    while not eof(f1) do                 { do for all of input file }
    begin
      i:=1;
      readln(f1,in_line);                { Get line from file and tack on end char }
      in_line1:=in_line;
      in_line:=in_line+chr(1);

      write(':');
      tot:=tot+1;                        { Give user something to look at }
      if (tot mod 70)=0 then writeln;

      case current_status^.kind of

{ This is where the information from the first pass is used.  If there was not
  an error in the first pass, the record corresponding to the current line
  will contain a 0.  If the first line was a comment or a blank line, the
  value will be a -1.  If the value is neither of these, the value is the
  error number and will be used to display the proper error message.         }

      -1 : writeln(f2,'                          | ',in_line1); { comment or blank line }

      1..9 : begin
                error_type:=current_status^.kind;    { Error in first pass }
                print_error;
             end;                                { errors }
```

C 25

```
      0 : begin
              parse_line;                        { parse line again }

          case opcode of

{ These Pseudo-ops are treated together, since they all require an output to
  the .REL file.                                                             }

                  org,nmi,irq,rst :
                        begin
                          if opcode=org then
                          begin
                            pc:=get_value;          { Calculate new PC }
                            result:=conhex(pc);
                          end
                          else result:=conhex(get_value);
                          writeln(f2,result,'                | ',in_line1);

{ These are the routines that write to the .REL file }

                          case opcode of
                                  nmi : begin
                                            writeln(f3);
                                            writeln(f3,'> FFFA');
                                          writeln(f3,copy(result,3,2),copy(result,1,2))
                                            end;

                                  irq : begin
                                            writeln(f3);
                                            writeln(f3,'> FFFE');
                                          writeln(f3,copy(result,3,2),copy(result,1,2));
                                            end;

                                  rst : begin
                                            writeln(f3);
                                            writeln(f3,'> FFFC');
                                          writeln(f3,copy(result,3,2),copy(result,1,2));
                                            end;

                                  org : begin
                                            writeln(f3);
                                            writeln(f3,'> ',result);
                                            end;

                          end;
                          length_line:=0;
                        end;                        {org}

{ No calculations needed for second pass equ statement }

          equ : writeln(f2,conhex(pc),'                | ',in_line1);

          ent,equ : ;          { No second pass work here either }
```

```
{ For both asc and byt, on this pass we need to not only update the PC
  properly, but also calculate what it is that is to be put out to the
  output file.  These are both messy routines during this pass.  For both of
  them the proper pair or single ascii characters must be picked out and then
  converted into the proper hex byte value.                                }

            asc : begin
                      j:=1;
                      writeln(f2,conhex(pc),'          | ',in_line1);
                      long_string:='';
                      while operand[j]<>'"' do
                      begin
                        long_string:=long_string+
                                    copy(conhex(ord(operand[j])),3,2)+' ';
                        j:=j+1;
                      end;                  {operand[j]}
                      pc:=pc+j-1;
                      print_spc(long_string);
                `   end;                      {asc}

            byt : begin
                      j:=0;
                      writeln(f2,conhex(pc),'          | ',in_line1);
                      long_string:='';
                      k:=pos(chr(1),operand);
                      insert(',',operand,k);
                      while operand[1]<>chr(1) do
                      begin
                        k:=pos(',',operand);   { each byte seperated by comma}
                        long_string:=long_string+
                       copy(conhex(make_number(copy(operand,1,k-1))),3,2)+' ';
                        delete(operand,1,k);
                        j:=j+1;
                      end;                    {operand[1]}
                      pc:=pc+j;
                      print_spc(long_string);
                  end;                        {byt}

        else
          parse_operand;    { Extract real operand }

          calc_opcode;      { Calculate the opcode value }

          calc_operand;     { Calculate the second and third bytes of inst}

          writeln(f2,conhex(pc),' ',lead_in+' | '+in_line1);
```

```
{ This prints to the output file .REL depending on how many bytes were in the
  instruction.                                                              }
                    case byte_no of
                      1 : begin
                              print_rel(op_code);
                              pc:=pc+1.0;
                            end;                    {one}

                      2 : begin
                              print_rel(op_code);
                              print_rel(operand1);
                              pc:=pc+2.0;
                            end;                    {two}

                      3 : begin
                              print_rel(op_code);
                              print_rel(operand1);
                              print_rel(operand2);
                              pc:=pc+3.0;
                            end;                    {three}
                    end;                            {case byte_no}
                  end;                              {case opcode}
                end;                                {no error}
            end;                                    {case current_status}
    if current_status^.next<>nil then current_status:=current_status^.next;
    end;                                            {pass two}

  writeln(f3);
  writeln(f3,'<');   { "End of file" marker for .REL file }
  writeln(f3);

  output_symbol_table;   { output symbol table to screen and .LST file }

  writeln(f2);
  writeln;
  writeln(f2,'total number of errors : ',errors);
  writeln('total number of errors : ',errors);

  close(f2);
  close(f3);
end.                                                {file}
```

```
;****************************************************************************
;
;                       MONITOR PROGRAM FOR 6502 SYSTEM
;
;                          WRITTEN BY MATTHEW FARRENS
;                          LAST MODIFIED 11/28/84
;
;    For proper monitor operation, following minimum memory requirements
;    must be met:
;
;    RAM:                    0000-01FF
;    EPROM (MONITOR):        F800-FBFF (not including TEST and RESULT programs)
;
;    SERIAL DEVICE:          C000 (assumes 6850 UART)
;    PTM         :           E000 (to provide transmission clocks to UART)
;
;    MONITOR resides in entire 1 K section from F800-FC00H, and uses locations
;    00E0-0110 in RAM.
;
;****************************************************************************


BUFF:    EQU 0E0H
ADRS:    EQU 0E2H
LOAD:    EQU 0E4H
CHECK:   EQU 0E5H
MSGS1:   EQU 0E7H             ; MEMORY LOCATIONS USED BY MONITOR
BRK:     EQU 0E9H
BRKADR:  EQU 0EAH
STRPOS:  EQU 0ECH
RESET:   EQU 0EDH



EX:      EQU 0F0H
WHY:     EQU 0F1H
EH:      EQU 0F2H
STATUS:  EQU 0F3H             ; LOCATIONS USED BY BREAK ROUTINE
SPNT:    EQU 0F4H
LOBYTE:  EQU 0F5H
HIBYTE:  EQU 0F6H

MSG:     EQU 0F8H

CUART:   EQU 0C000H
DUART:   EQU 0C001H

PTM      EQU 0E000H                    ; PROGRAMMABLE TIMER MODULE
```

```
         ORG OF800H                    ; START MONITOR AT BEGINNING OF EPROM

         BYT ODH,OAH,OAH

         ASC "         6502 DEVELOPMENT SYSTEM"
         BYT ODH,OAH,OAH
         ASC "L      - Download from external source"
         BYT ODH,OAH
         ASC "Mxxxx - Examine memory location xxxx"
         BYT ODH,OAH
         ASC "Gxxxx - Begin execution at location xxxx"
         BYT ODH,OAH
         ASC "Bxxxx - Breakpoint at location xxxx"
         BYT ODH,OAH
         ASC "C      - Continue from last breakpoint"
         BYT ODH,OAH
         ASC "R      - Display register contents"
         BYT ODH,OAH
         ASC "H      - This message"
         BYT ODH,OAH,OAH

         ASC "Added features"

         BYT ODH,OAH,OAH

         ASC "T      - Execute remote memory test"
         BYT ODH,OAH
         ASC "P      - Print out results of test"

         BYT ODH,OAH,OAH,00
REGS:    BYT 58H,00,59H,00,41H,00,53H,00,53H
         BYT 50H,00,50H,43H,4CH,00,50H,43H,48H,00

STRING:  BYT ODH,20H,5BH,OAH,ODH

BRKMSG:  BYT ODH,OAH
         ASC "Hit Breakpoint"
         BYT ODH,OAH,00
```

```
;     ------------ ROUTINE TO HANDLE SOFTWARE INTERRUPT (BRK)  ----------------

;         On execution of software break instruction (00 hex) program execution
;         begins here, since 6502 treats BRK as a kind of IRQ.  Here registers,
;         status reg, stack pointer and program counter at time of break are
;         saved.  Due to way Break is handled, program counter must be
;         decremented by two before saving.

BREAK:  STX EX,0
        STY WHY,0          ; SAVE REGISTER VALUES
        STA EH,0

        PLA
        STA STATUS,0       ; SAVE STATUS REG

        PLA
        STA LOBYTE,0       ; SAVE LOW BYTE OF PROGRAM COUNTER

        PLA
        STA HIBYTE,0       ; SAVE HIGH BYTE OF PROGRAM COUNTER

        TSX
        STX SPNT,0         ; SAVE THE STACK POINTER

        DEC LOBYTE,0
        BNE BREAK1
        DEC HIBYTE,0       ; ADJUST FOR WAY BRK IS HANDLED
BREAK1: DEC LOBYTE,0

        LDX #0EH
        TXS                ; POINT STACK AT WHERE WE WANT IT

        LDX #0
        LDA BRK,0          ; PICK UP WHAT WAS AT LOCATION
        STA (BRKADR,X)     ; PUT IT BACK WHERE IT CAME FROM

BREAK2: LDA BRKMSG,X
        BEQ START          ; SEND MESSAGE THAT BREAKPOINT HAS BEEN HIT
        JSR SEND
        INX
        BNE BREAK2
```

```
; --------------------------- RESET ENTRY POINT ----------------------------
;      .    Program comes here on reset.  UARTs are reset and initialized,
;           and stack pointer is set to very lowest part of the stack.
;           If we have been here once, the introductory message is not printed.
;           Otherwise, the welcoming message is produced.
;           Save regs and stack pointer, cannot access program counter

OPEN:    STA EH,0
         STX EX,0
         STY WHY,0        ; SAVE REG VALUES

         PHP
         PLA
         STA STATUS,0     ; SAVE STATUS REG

         TSX
         STX SPNT,0       ; SAVE STACK POINTER

         LDA #83H         ; FIRST CONTROL 2 REG
         STA PTM+1

         LDA #0
         STA PTM          ; INITIALIZE THE PTM
         STA PTM+2

         LDA #25
         STA PTM+5        ; SET UP THE PTM FOR COMMUNICATIONS

         LDA #03
         STA CUART        ; RESET UART

         LDA #01
         STA CUART        ; 7 BITS 2 STOP BITS EVEN PARITY 1200 BAUD

         LDX #0EH
         TXS              ; MONITOR ONLY USES BOTTOM OF STACK

         LDX #0
         STX BRK,0        ; CLEAR BREAK HOLDER

         LDX RESET,0
         CPX #0EDH
         BEQ START        ; IF WE HAVE BEEN HERE ONCE, SKIP MESSAGE
         LDX #0EDH
         STX RESET,0

OPEN1:   LDX #MSG
         STX MSGS1+1,0
         LDX #0           ; SET UP MEMORY FOR DISPLAYING OF MESSAGE
         STX MSGS1,0
```

D 4

```
INTRO:  LDA (MSGS1,X)
        BEQ START        ; NULL (OO) IS END OF STRING
        JSR SEND         ; ELSE SEND VALUE

        INC MSGS1,O
        BNE INTRO
        INC MSGS1+1,O    ; DO UNTIL DONE
        BNE INTRO

;       --------------------    STANDARD ENTRY POINT    --------------------------
;
;       This is where the prompt is printed and the keystroke is decoded
;       and interpreted.  If not one of the legal commands, print prompt
;       again and start over.

START:  LDX #04
        LDA STRING,X
        JSR SEND         ; SEND SIGN ON STRING
        DEX
        BNE START+2

        LDA #O
        STA LOAD,O       ; CLEAR LOAD FLAG INDICATING WE ARE NOT IN DOWNLOAD

        JSR ECHO         ; GET CHAR FROM TERMINAL AND SEND IT BACK OUT

        CMP #47H
        BEQ GO           ; "G" IS GO (RUN PROGRAM)

        CMP #4DH
        BEQ EXAM         ; "M" IS EXAMINE MEMORY

        CMP #48H
        BEQ OPEN1        ; "H" IS SEND HELP MESSAGE

        CMP #42H
        BEQ BRKPT        ; "B" IS ENTER BREAKPOINT

        CMP #43H
        BEQ CONT         ; "C" IS CONTINUE FROM LAST BREAKPOINT

        CMP #52H
        BEQ REG          ; "R" IS DISPLAY REGISTER VALUES AT LAST BREAK

        CMP #54H
        BNE MAIN1
        JMP MEMTST       ; "T" IS RUN MEMORY TEST PROGRAM (OPTIONAL)

MAIN1:  CMP #50H
        BNE MAIN2
        JMP RESULT       ; "P" IS PRINT RESULTS PROGRAM (ALSO OPTIONAL)

MAIN2:  CMP #4CH         ; "L" IS LOAD PROGRAM
        BNE START        ; IF NONE OF ABOVE, INVALID COMMAND
        JMP DNLOAD
```

D 5

```
;      ------------------    TRANSFER CONTROL ROUTINE   -------------------------

;         This routine gets the address to start executing at and transfers
;         control to that location after resetting the stack pointer to
;         the top where it belongs. and loading regs.

GO:       JSR ADLOAD         ; GET THE ADDRESS TO TRANSFER CONTROL TO

          LDX #0FFH
          TXS                ; PUT STACK POINTER BACK AT TOP

          LDA ADRS+1,0
          PHA                ; PUSH HIGH ADDRESS ONTO STACK

          LDA ADRS,0
          PHA                ; PUSH LOW BYTE OF ADDRESS ONTO STACK

          JMP CONT1          ; PUSH STUFF AND START EXECUTING




;      -------------   ROUTINE TO CONTINUE FROM LAST BREAKPOINT    ----------------

;         This routine allows the program to resume exection from where the
;         last breakpoint occured.  It restores all the registers and the
;         stack pointer and transfers control to the original location.

CONT:     LDX SPNT,0
          TXS                ; PUT STACK POINTER BACK WHERE IT WAS

          LDX #0
          STX BRK,0          ; CLEAR BREAK LOCATION

          LDA HIBYTE,0
          PHA                ; PUSH HIGH BYTE ON STACK

          LDA LOBYTE,0
          PHA                ; PUSH LOW BYTE ON STACK

CONT1:    LDA STATUS,0
          PHA                ; PUSH STATUS REG ON STACK

          LDA EH,0
          LDX EX,0
          LDY WHY,0          ; RESTORE REGISTERS

          RTI                ; PICK UP WHERE WE LEFT OFF
```

```
; -------------------- ROUTINE TO EXAMINE MEMORY --------------------

;       This routine displays the contents of the requested memory location
;       and allows the user to change the value stored at that location if
;       so desired.

EXAM:   JSR ADLOAD        ; GET ADDRESS TO DISPLAY

        LDA #20H
        JSR SEND          ; PRINT SPACE

EXAM1:  JSR PRNT          ; DISPLAYS WHAT IS STORED AT ADLOAD AND ALLOWS USER
        JMP EXAM1         ; TO CHANGE VALUE IF NEEDED.




; -------------------- ROUTINE TO ENTER BREAKPOINT --------------------

;       This routine is where a breakpoint is entered.  The value that
;       was stored at the desired location is saved so that it can be
;       restored on execution of the Break.

BRKPT:  JSR ADLOAD        ; GET DESIRED ADDRESS OF BREAK

        LDA ADRS,0
        STA BRKADR,0      ; STORE WHERE BREAK IS
        LDA ADRS+1,0
        STA BRKADR+1,0

        LDX #0
        LDA (ADRS,X)
        STA BRK,0         ; STORE WHAT WAS THERE

        TXA
        STA (ADRS,X)      ; PUT BREAK IN AT PROPER LOCATION

        JMP START         ; DONE WITH ROUTINE, START AGAIN
```

D 7

```
;    ----------- ROUTINE TO DISPLAY LAST CONTENTS OF REGISTERS    -------------

;          This routine displays the saved values of the registers and
;          the stack pointer and the program counter at the time of the
;          last software Break instruction.  Simply running this routine
;          without a Break instruction being executed first will not give
;          an accurate picture of the registers at the time of the command.

REG:       LDA #EX           ; PICK UP WHERE REGISTERS ARE STORED
           STA ADRS,O

           LDA #0
           STA ADRS+1,O      ; SET UP MEMORY CONTENTS FOR SUBROUTINES
           STA STRPOS,O      ; CLEAR STRING COUNTER

           LDA #20H
           JSR SEND          ; SEND SPACE

REGO:      LDX STRPOS,O
REG1:      LDA REGS,X
           BEQ REG2          ; PICK UP VALUE TO SEND, SEND UNTIL NULL
           JSR SEND
           INX
           BNE REG1

REG2:      INX
           STX STRPOS,O      ; POINT AT NEXT STRING TO TRANSMIT

           LDA #3DH
           JSR SEND          ; SEND THE =

           JSR PRNT          ; SEND VALUE OF REGISTER AND CHANGE IF DESIRED

           LDA #HIBYTE
           CMP ADRS,O        ; SEE IF WE HAVE DONE ALL YET
           BPL REGO          ; DO FOR ALL REGS

           JMP START         ; ALL DONE WITH REGS, START AGAIN
```

D 8

```
;    ---------------- DOWNLOAD FROM EXTERNAL SOURCE ROUTINE ------------------
;         This routine reads from the Serial Port and loads into memory
;         as determined by commands imbedded in the incoming data.
;         This is definitely NOT INTEL Hex format.  There is no formal
;         parity checking.  In this routine, upon receipt of a ">" the
;         program interprets the next 4 ascii characters as a memory
;         location at which to begin the storing of data.  "<" is the
;         "end of file" marker.  This routine is left when the "<" is
;         received and control is returned to the monitor.

DNLOAD: DEC LOAD,0       ; SET NO ECHO FLAG, DON'T ECHO TO INCOMING FILE

TMP5:   JSR ECHO         ; GET CHAR
        CMP #3EH         ; COMPARE TO >
        BNE TMP5         ; WAIT UNTIL IT COMES IN

TMP6:   JSR ADLOAD       ; GET THE 16 BIT ADDRESS

TMP7:   LDX #2
        STX CHECK,0      ; WE ARE WAITING FOR 2 ASCII CHARS

GETR:   JSR ECHO         ; GET CHAR
        CMP #3EH         ; COMPARE TO >
        BEQ TMP6

        CMP #3CH         ; COMPARE TO <
        BNE TMP0
        JMP START        ; IF SO, ALL DONE AND START OVER

TMP0:   JSR VALID        ; SEE IF INCOMING CHAR IS VALID HEX
        BNE TMP7         ; IF NOT, GET NEW STUFF

        DEC CHECK,0
        BEQ TMP8         ; IF VALID, SEE IF WE ARE IN MIDDLE OF NIBBLES

        ASL
        ASL
        ASL
        ASL              ; SHIFT INTO HIGH NIBBLE
        STA BUFF,0       ; STORE TEMPORARILY
        JMP GETR         ; GET SECOND HALF OF BYTE (NEXT CHAR)

TMP8:   ORA BUFF,0       ; PICK UP HIGH NIBBLE
        LDX #0           ; CLEAR X
        STA (ADRS,X)     ; STORE IN PROPER PLACE

        INC ADRS,0       ; INCREMENT LOW BYTE
        BNE TMP9

        INC ADRS+1,0     ; INCREMENT HIGH BYTE

TMP9:   JMP TMP7         ; GO BACK AND GET NEXT THING
```

D 9

```
;    ============================= SUBROUTINES  =============================

;    ::::::::  SUBROUTINE TO DISPLAY LOCATION & GET NEW ONE IF THERE  :::::::::::

;         This is a heavy duty subroutine.  It calls several others in the
;         line of duty.  This subroutine first sends out the value of the
;         memory location stored at addr,addr+1. Then it sends a delimitering
;         space, and checks the incoming char.  If the incoming char is also
;         a space, that indicates that the next sequential memory location is
;         to be displayed.  If the incoming char is a valid hex digit, it is
;         assumed that the value stored at addr,addr+1 is to be replaced by
;         the incoming value.  If the incoming char is a nonvalid hex digit,
;         it assumes you screwed up and returns you to the main monitor.

;         REGS DESTROYED:          X,Y,A

PRNT:   JSR DSPLY        ; DISPLAY THE VALUE AT THE MEMORY LOCATION

        LDA #20H
        JSR SEND         ; SEND DELIMITERING SPACE

        JSR ECHO         ; GET AND ECHO CHAR

        CMP #20H
        BEQ NEXTO        ; IF SPACE, DISPLAY NEXT MEMORY LOCATION

        LDX #1
        JSR GOTONE       ; IF VALID HEX, WE WILL RETURN, ELSE ABORT

        LDX #00
        STA (ADRS,X)     ; STORE NEW VALUE IN MEMORY LOCATION

        LDA #20H
        JSR SEND         ; SEND SPACE

        JMP INK          ; GO TO NEXT MEMORY LOCATION

NEXTO:  LDA #08
        JSR SEND         ; SEND A BACKSPACE CHAR

INK:    INC ADRS,0       ; INC LOW BYTE
        BNE PRNT1

        INC ADRS+1,0     ; IF ROLLOVER, INC HIGH BYTE

PRNT1:  RTS
```

D 10

```
;   ::::::::::::::   GET TWO ASCII CHARS, PRODUCE SINGLE BYTE   ::::::::::::::::::

;       Routine to take two ascii chars and make them into a singe 8bit
;       byte.

;       REGS DESTROYED:         X,Y,A

GETBYT: LDX #1          ; SET COUNTER

TMP2:   JSR ECHO        ; GET CHAR AND ECHO

GOTONE: JSR VALID       ; SEE IF VALID HEX DIGIT
        BEQ TMP4        ; IF NOT, SEE WHICH ROUTINE CALLED

        LDX LOAD,0
        BEQ OUT         ; IF NOT LOADING FROM FILE, ERROR

        CMP #20H        ; SEE IF IT IS A SPACE
        BEQ GETBYT      ; WAIT FOR FIRST VALID CHAR AFTER >

OUT:    LDX #0EH        ; SET X TO TOP OF MONITOR STACK
        TXS             ; SET STACK POINTER BACK TO TOP OF STACK
        JMP START       ; START OVER DUE TO ERROR.

TMP4:   STA BUFF,X,0    ; STORE IN BUFFER

        DEX
        BPL TMP2        ; DO FOR BOTH BYTES

        LDA BUFF+1,0    ; PICK UP FIRST BYTE
        ASL
        ASL
        ASL
        ASL             ; MOVE INTO HIGH NIBBLE
        ORA BUFF,0      ; OR IN LOW NIBBLE
        STA BUFF,0      ; STORE IN BUFFER.

        RTS
```

D 11

```
;   :::::::::::::::::    SEE IF CHAR IN A IS VALID HEX CHAR    ::::::::::::::::::::

;           Subroutine to check validity of hex char.  If char is not in hex
;           bounds then Y reg is set to minus one.  If incoming char is in
;           legal hex char set, then Y reg is set to zero.

;           REGS DESTROYED:            Y,A

VALID:  CMP #47H            ; TEST FOR GREATER THAN F.
        BPL NOGOOD

        CMP #30H            ; TEST FOR LESS THAN 0.
        BMI NOGOOD

        CMP #3AH            ; TEST FOR DIGIT.
        BMI OK

        CMP #41H
        BPL FIX
        BMI NOGOOD          ; IF BETWEEN 3A AND 41, WRONG ELSE MAKE ASCII

FIX:    CLD
        SEC
        SBC #07             ; MAKE LETTER VALID HEX DIGIT.

OK:     AND #0FH
        LDY #00
        RTS                 ; CLEAR OFF TOP END, SET Y AND LEAVE

NOGOOD: LDY #-1
        RTS                 ; SET Y TO INVALID, RETURN.




;   :::::::::::::::::    LOAD 4 ASCII CHARS INTO 16 BIT ADDR    ::::::::::::::::::::

;           This is another subroutine that calls others.  This subroutine
;           gets four valid ascii characters and stores them as an address,
;           in the standard 6502 form of low byte followed by high byte.
;           However, it assumes the address is coming in "right" (i.e. high
;           byte followed by low byte).

;           REGS DESTROYED:            X,Y,A

ADLOAD: JSR GETBYT          ; GET HIGH BYTE OF ADDRESS
        STA ADRS+1,0        ; STORE IN HIGH BYTE

        JSR GETBYT          ; GET LOW BYTE OF ADDRESS
        STA ADRS,0          ; STORE IN LOW BYTE

        RTS                 ; ALL DONE
```

D 12

```
;   :::::::::::::   SUBROUTINE TO DISPLAY BYTE AS TWO ASCII CHARS   :::::::::::::::

;        This subroutine actually has two entry points.  The monitor always
;        enters at DSPLY, since it always wants at least two consecutive
;        ascii characters sent at the same time.  However,  the test program
;        enters at DSPLY1 since it only wants to send characters one at a time.

;        REGS DESTROYED:          X,A

DSPLY:  LDX #00
        LDA (ADRS,X)      ; PICK UP THE BYTE

DSPLY1: PHA               ; SAVE ON STACK
        LSR
        LSR
        LSR
        LSR
        JSR ASKIZ         ; ASCIIIZE THE HIGH NIBBLE

        PLA               ; GET ORIGINAL VALUE BACK
        AND #0FH          ; CLEAR OFF TOP NIBBLE
        JSR ASKIZ         ; ASCIIIZE THE LOW NIBBLE

        JSR CHKIN         ; SEE IF CHARACTER HAS COME IN

        RTS



;   :::::::::::::::::::   SUBROUTINE TO MAKE NIBBLE ASCII   :::::::::::::::::::::

;        This subroutine converts the value stored in the A register into
;        a valid ascii character.

;        REGS DESTROYED:          A

ASKIZ:  ORA #30H          ; SET ASCII BIT

        CMP #3AH          ; SEE IF LETTER
        BMI SEND          ; IF NOT, RETURN

        CLC
        ADC #07           ; IF SO, ADD 7
```

D 13

```
;    :::::::::::::::::::::  SEND CHARACTER IN A REG    :::::::::::::::::::::::::::

;          This routine sends out whatever is in the A reg to the UART.
;          It waits (unnecessarily?) for the UART to finish the last
;          transmission before starting a new one.

;          REGS DESTROYED:          NONE

SEND:   PHA                ; SAVE A REG

        LDA CUART
        AND #2
        BEQ SEND+1         ; STANDARD WAIT UNTIL TXBUF EMPTY

        PLA
        STA DUART          ; SEND IT

        RTS



;    :::::::::::::::::::::  CHECK FOR INCOMING CHARACTER    ::::::::::::::::::::::::::

;          This routine checks for an incoming character.  If there is
;          one it checks to see if it is a control S (suspend transmission).
;          If it is, we wait here until the control Q comes in allowing us to
;          resume transmission.  If a character is waiting, we load it into
;          the A reg and set all bits high in X.  If nothing is waiting X
;          remains all clear.

;          REGS DESTROYED:          A,X

CHKIN:  LDA CUART
        AND #1
        BEQ CHKIN1         ; IF NO CHAR, THEN RETURN

        LDX #0FFH          ; SET ALL BITS HIGH INDICATING CHAR BEEN RECEIVED

        LDA DUART
        CMP #13H           ; CHECK FOR CONTROL S
        BNE CHKIN1

CHKIN2: LDA CUART
        AND #1             ; WAIT FOR INCOMING CONTROL Q
        BEQ CHKIN2

        LDA DUART
        CMP #11H           ; IF NOT CONTROL Q, WAIT
        BNE CHKIN2

        LDX #0             ; IF ^S ^Q PAIR, PRETEND NOTHING HAPPENED

CHKIN1: RTS
```

```
;     :::::::::::::::::::     ECHO CHAR, IF NOT IN LOAD MODE     :::::::::::::::::::::

;          Subroutine to get character from UART, save it, and echo the char
;          back out to the UART so the sender knows it was received.

;          REGS DESTROYED:               A

ECHO:     LDA CUART
          AND #1
          BEQ ECHO              ; STANDARD WAIT FOR CHAR

          LDA DUART             ; GET CHAR
          AND #7FH              ; CLEAR OFF TOP BIT

          PHA                   ; SAVE CHAR ON STACK
          LDA LOAD,O
          BEQ SEND+1            ; IF WE ARE ECHOING, SEND CHAR

          PLA                   ; IF WE ARE LOADING, DO NOT SEND BACK OUT.
          RTS                   ; POP A OFF TOP OF STACK
;****************************************************************************************
;
;                              End of Monitor Program
;
;****************************************************************************************
```

D 15

MAIN MONITOR PROGRAM
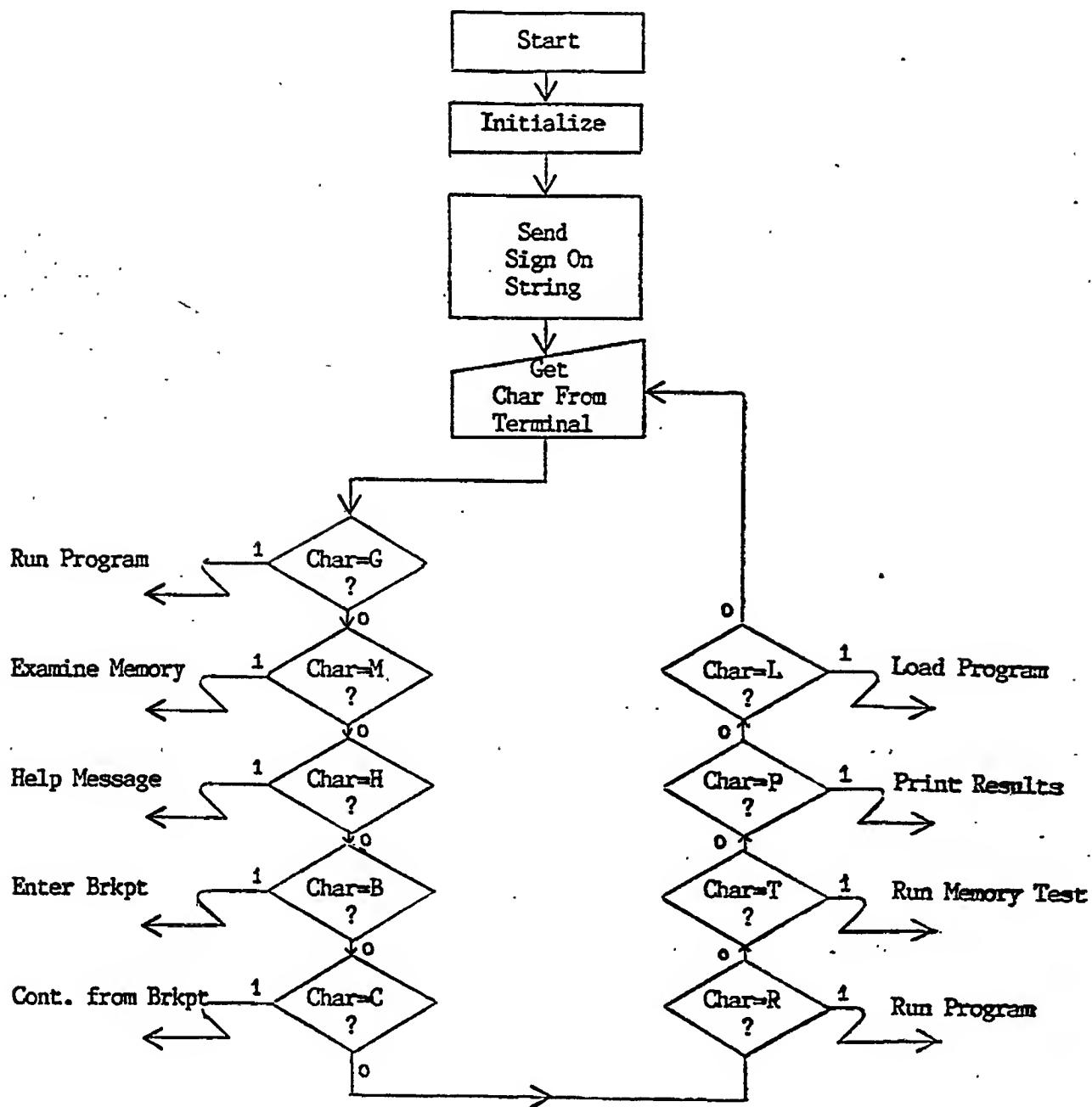
SOFTWARE INTERRUPT ROUTINE

```
interrupt  ──────⌐
                 └──→    ┌──────────────┐
                         │     Save     │
                         │   Register   │
                         │    Values    │
                         └──────────────┘
                                │
                                ↓
                         ┌──────────────┐
                         │     Save     │
                         │    Status    │
                         │   Register   │
                         └──────────────┘
                                │
                                ↓
                         ┌──────────────┐
                         │  Save Low    │
                         │  and High    │
                         │  Bytes of    │
                         │   the PC     │
                         └──────────────┘
                                │
                                ↓
                         ┌──────────────┐
                         │     Save     │
                         │    Stack     │
                         │   Pointer    │
                         └──────────────┘
                                │
                                ↓
                         ┌──────────────┐
                         │  Point Stack │
                         │   Pointer    │
                         │  at Monitor  │
                         │    Stack     │
                         └──────────────┘
                                │
                                ↓
                         ┌──────────────┐
                         │   Pick up    │
                         │   Contents   │
                         │  of Brkpt    │
                         │   Address    │
                         └──────────────┘
                                │
                                ↓
                         ┌──────────────┐
                         │   Restore    │
                         │   Contents   │
                         │  at Brkpt    │
                         │   Address    │
                         └──────────────┘
                                │
                                ↓
                         ╱──────────────╲
                        ╱   Send Msg.    │
                        ╲  that Break     │
                         ╲   was hit     ╱
                          ╲─────────────╱
                                │          Return to START
                                ↓
```

E 2

RESET ENTRY ROUTINE

```
┌──────────────┐
│     Save     │
│   Register   │
│    Values    │
└──────────────┘
        │
        ∨
┌──────────────┐
│     Save     │
│    Status    │
│   Register   │
└──────────────┘
        │
        ∨
┌──────────────┐
│     Save     │
│    Stack     │
│   Pointer    │
└──────────────┘
        │
        ∨
┌──────────────┐
│  Initialize  │
│  PTM set up  │
│ for commun-  │
│   ication    │
└──────────────┘
        │
        ∨
┌──────────────┐
│    Reset     │
│    UART      │
└──────────────┘
        │
        ∨
┌──────────────┐
│    Clear     │
│    Break     │
│    Holder    │
└──────────────┘
        │
        ∨
      ◇ Reset ◇              ┌──────────────┐
     ◇ Before  ◇    0        │    Setup     │
     ◇    ?    ◇ ────────>   │   Memory     │
      ◇       ◇              │ For Message  │
         │                   │   Display    │
         │ 1                 └──────────────┘
         │                          │
         └──────────────┬───────────┘
                        ∨
                Return to START
```

E 3

G - RUN A PROGRAM

```
Char=G
  ?
```

```
    Get
Address for
  Control
 Transfer
```

```
  Push
 Address
on Stack
```

```
 Transfer
Control to
 Interrupt
Address via
Return from
 Interrupt
Instruction
```

Return to START

M - EXAMINE MEMORY

```
   Char=M                Get Hi
     ?      1            Byte &
                         Store it
                            │
                            ▼
                         Get Low
                         Byte &
                         Store it
                            │
                            ▼
                          Print
                          Space
                            │
                            ▼
                         Display
                         Value in
                         Memory
                            │
                            ▼
                         Get Chars
                         From
                         Terminal
                            │
                            ▼
          Char=" "    1     Send          Increment
            ?               Backspace     Memory
                            Character     Location
            │0
            ▼
  Put New      1    Is
  Chars in          Char valid
  Memory            Hex Char
                       │0

              Return to START
```

H-HELP MESSAGES

Char=H ?   1

Set up
Memory
with
Messages

Get a
Char

Char=null ?   o   →   Display Char

Return to START

1

B-ENTER A BREAKPOINT

```
  ◇ Char=B      1
    ?     ────────⌇─────►

              ┌──────────────┐
              │     Get      │
              │   Breakpt    │
              │   Address    │
              └──────┬───────┘
                     │
                     ▼
              ┌──────────────┐
              │    Store     │
              │  Address in  │
              │ Accumulator  │
              └──────┬───────┘
                     │
                     ▼
              ┌──────────────┐
              │    Store     │
              │   Contents   │
              │  of Breakpt  │
              │   Address    │
              └──────┬───────┘
                     │
                     ▼
              ┌──────────────┐
              │    Insert    │
              │   Breakpt    │
              └──────┬───────┘
                     │        Return to START
                     ⌇
                     ▼
```

C-CONTINUE FROM BREAKPOINT

```
         ╱╲
        ╱  ╲
       ╱Char=C╲  1
       ╲  ?   ╱────────►
        ╲    ╱
         ╲  ╱
          ╲╱
```

┌──────────────┐
│   Restore   │
│   Stack   │
│   Pointer   │
└──────────────┘
        │
        ▼
┌──────────────┐
│   Clear   │
│   Break   │
│   Location   │
└──────────────┘
        │
        ▼
┌──────────────┐
│ Push Stutus │
│   Register   │
│   on Stack   │
└──────────────┘
        │
        ▼
┌──────────────┐
│   Restore   │
│ Registers   │
└──────────────┘
        │
        ▼
        Return to user's
        interrupted program
        │
        ▼

R - DISPLAY REGISTERS

Char=R ?

Get Address of Registers

Send a Space

Pick up Character

Char=Null ?

Display Char

Display "=" sign

Display Register Contents

Increment Memory Address

All Registers Printed

Return to START

E9
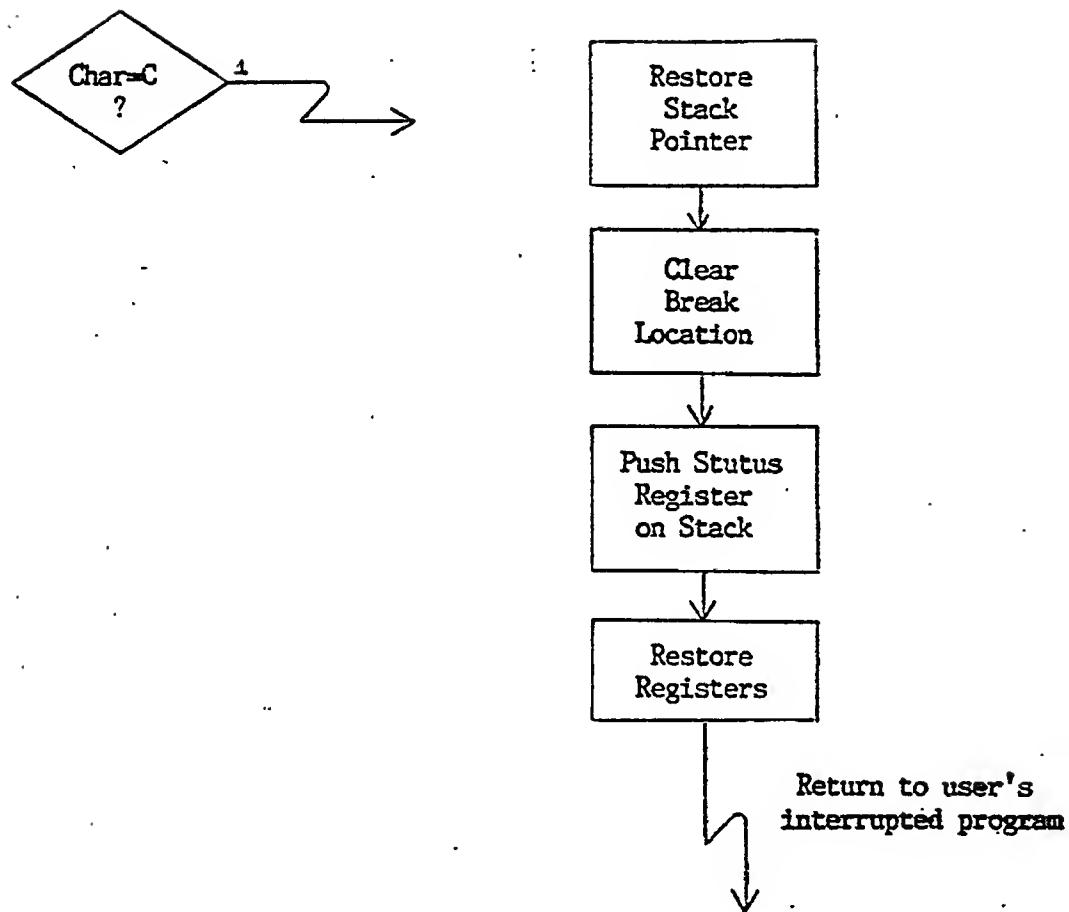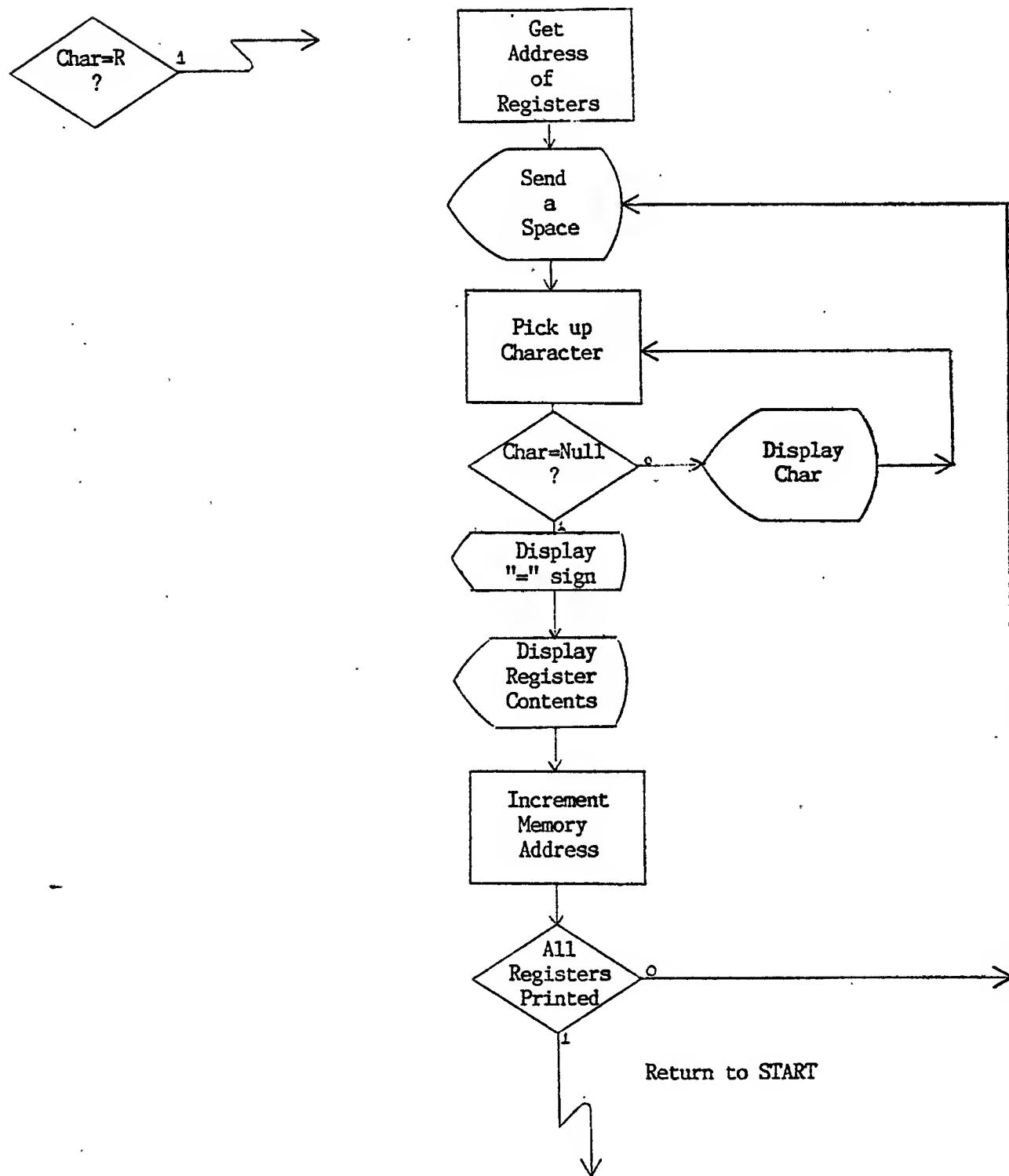
T- MEMORY TEST PROGRAM

P-PRINT RESULTS OF MEMORY TEST

```
      ┌──────────┐
      │  Char=P  │ ────────────╲──────►
      │    ?     │
      └──────────┘
```

```
                    ┌──────────────┐
                    │  Get Error   │
                    │ Buffer Pntr  │
                    │   Address    │
                    └──────────────┘
                           │
                           ▼
                    ╱──────────────╲
                    │   Display     │
                    │  Experiment   │
                    │  Start time   │
                    ╲──────────────╱
                           │
                           ▼
                    ◇──────────────◇        1
          ┌────────►│     Done     │ ──────────╲──────►   Return to Monitor
          │         │      ?       │
          │         ◇──────────────◇
          │                │ 0
          │                ▼
          │         ┌──────────────┐
          │         │     Get      │
          │         │   Contents   │
          │         │  of Address  │
          │         └──────────────┘
          │                │
          │                ▼
          │         ╱──────────────╲
          │         │   Display     │
          │         │   Contents    │
          │         ╲──────────────╱
          │                │
          │                ▼
          │         ┌──────────────┐
          │         │  Increment   │
          │         │    Buffer    │
          │         │    Memory    │
          │         │   Address    │
          │         └──────────────┘
          │                │
          └────────────────┘
```

E 11
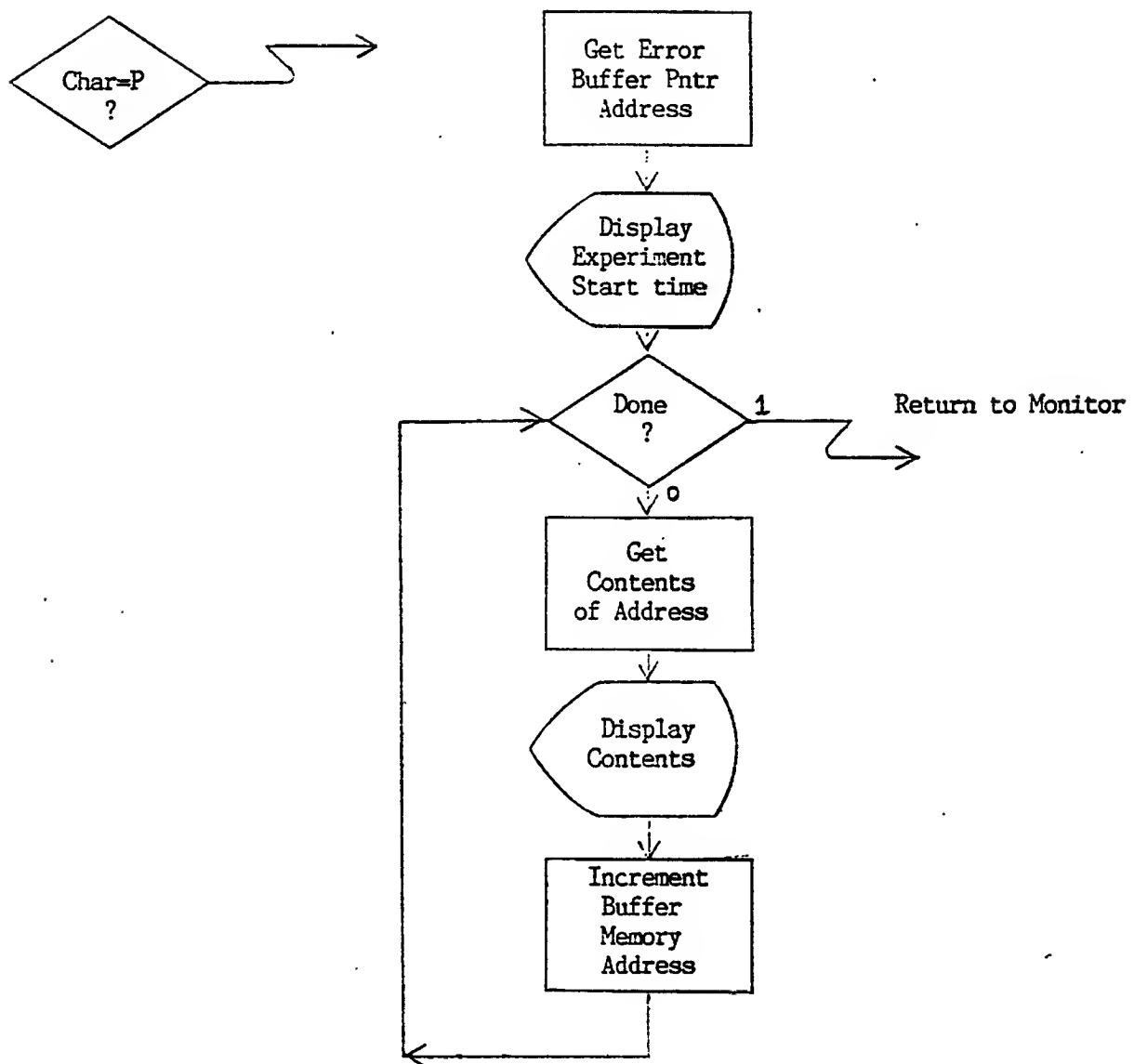
L - DOWNLOAD A PROGRAM

E 12
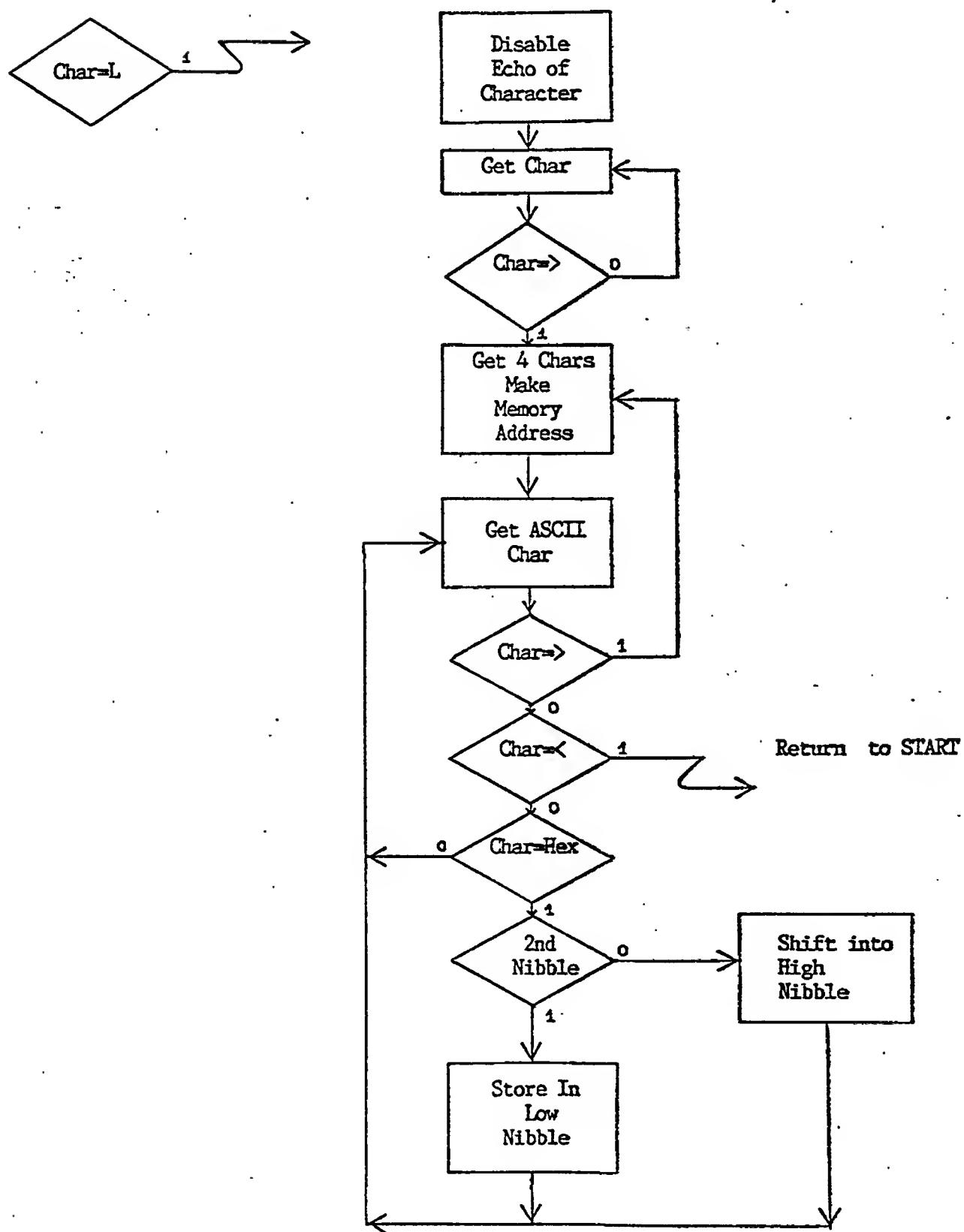
```
;****************************************************************************
;
;                        MEMORY TEST PROGRAM
;
;                     WRITTEN BY MATTHEW FARRENS
;
;                       LAST MODIFIED 12/8/84
;  ..
;
;
;       This is the Remote Memory testing program written to allow testing
;       of static RAM chips while they are undergoing irradiation.
;
;       The testing algorithm used is the Nair, Thatte and Abraham testing
;       procedure.
;
;****************************************************************************

;        Set up our programming environment

PIA1AP   EQU 0D800H
PIA1AC   EQU 0D801H
PIA1BP   EQU 0D802H           ; PIA ports for input and output
PIA1BC   EQU 0D803H

PIA2AP   EQU 0D000H
PIA2AC   EQU 0D001H
PIA2BP   EQU 0D002H
PIA2BC   EQU 0D003H

CUARTM   EQU 0C000H
DUARTM   EQU 0C001H           ; Uart for I/O

PTM1     EQU 0E000H           ; Timer module for keeping track of time into run

ADLOW    EQU 00               ; Used as a marker to clear low memory
ADCOUNT  EQU 03               ; Count of number of 256 byte pages per memory chip
SHFTCNT  EQU 04               ; Count of number of bits in byte
TYPE     EQU 05               ; Number of writes to memory to do (0,1,2)
MASK     EQU 06               ; Bit in the byte we are dealing with
DIRECT   EQU 07               ; Direction of memory traversal
ERCOUNT  EQU 08               ; Number of errors counted this time through
PASS     EQU 0AH              ; Number of times test has been executed
BUFFPT   EQU 0DH              ; Pointer to next available spot in error buffer
LASTPAS  EQU 0FH              ; Number of errors found during last pass through test
TIME     EQU 12H              ; Beginning time of memory test

BEGBUF   EQU 2
ENDBUF   EQU 10H              ; End of transmission buffer

;        PIA1AP = low 8 bits of address
;        PIA1BP = data: high nibble is input, low nibble is output
;        PIA2AP = control lines and upper address (WE⁻ CS⁻ OE⁻ X X X A9 A8)
;                                                  A7  A6   . . .      A0
```

F 1

```
;*******************************************************************************
;
;                          Program initialization
;
;*******************************************************************************

        ORG OFCOOH

MESS    ASC "Remote memory test program"
        BYT ODH,OAH,OAH

        ASC "Enter starting time of test : "
        BYT O

MEMTST  LDA #0
        STA PIA1AC
        STA PIA1BC          ; Talk to direction registers in PIAs
        STA PIA2AC

        LDA #OFFH
        STA PIA1AP          ; A ports are both output ports
        STA PIA2AP

        AND #OFH
        STA PIA1BP          ; B port has upper nibble inputs, lower outputs

        LDA #04
        STA PIA1AC
        STA PIA2AC          ; Now we want to talk to the ports themselves

        LDA #36H            ; CB2 output starts low
        STA PIA1BC
        STA PIA2BC          ; CB1 input set to trigger on rising edge of signal

        LDA #0
        STA PTM1+1          ; Start by talking to Control Register 3

        LDA #80H
        STA PTM1            ; Configure #3 to count on incoming signal

        LDA #83H
        STA PTM1+1          ; Configure #2 to count E clocks

        LDA #82H
        STA PTM1            ; Also configure #1 to count E clocks

        LDA #0
        STA PTM1+4
        LDX #25             ; Initialize #2 for UART use (52 uSec cycle time)
        STX PTM1+5

        STA PTM1+6
        LDX #24             ; Initialize #3 for 5 second cycle time
        STX PTM1+7
```

F 2

```
        LDA #0C3H
        STA PTM1+2
        LDA #4FH          ; Initialize #1 for 100 mSec cycle time
        STA PTM1+3

        LDX #13H
        LDA #0
LOOP    STA ADLOW,X,0     ; Clear low part of memory
        DEX
        BPL LOOP

        LDA #BEGBUF
        STA BUFFPT+1,0    ; Initialize BUFFPT to beginning of error buffer

        JSR CRLF          ; Prepare to send message

        LDX #0
MESOUT  LDA MESS,X
        BEQ DONE1         ; Send beginning message to terminal
        JSR SENDM
        INX
        BNE MESOUT

DONE1   JSR GETIME        ; Get starting time for reference

        JSR CRLF          ; Send carriage return-line feed
```

F 3

```
;****************************************************************************
;
;        Here we wait until 5 seconds has expired.  By doing this, we can
;        ensure that the number of times through the test * 5 is the number
;        of seconds into the run.  This is also the "entry" point into the
;        program. We jump to here after every pass.
;
;****************************************************************************
WAIT      LDA PIA2BC         ; Get the clock toggle
          BPL WAIT           ; Wait for 5 second timer
          LDA PIA2BP         ; When 5 seconds up, read port to clear timer flag



;****************************************************************************
;
;        The actual remote memory testing program sequence begins here.
;        The first 4 sequences provide the test for all stuck-at faults
;        in the memory array, decoder and memory address register
;        First, the entire memory array must be set to zero
;
;****************************************************************************

CLEAR     LDX #0             ; What is to be written out
          JSR INIT           ; Clear all of test memory
```

```
;----------------------------------------------------------------------

SEQ1    LDY #0
        LDA #1            ; Read low, write high from beginning to end
        STA TYPE,0
        JSR COUNTUP

        LDY #1
        LDA #0
        STA TYPE,0        ; Read high, write nothing from end to beginning
        JSR COUNTDN

;----------------------------------------------------------------------

SEQ2    LDY #1
        LDA #1
        STA TYPE,0        ; Read high, write low from beginning to end
        JSR COUNTUP

        LDY #0
        LDA #0
        STA TYPE,0        ; Read low, write nothing from end to beginning
        JSR COUNTDN

;----------------------------------------------------------------------

SEQ3    LDY #0
        LDA #1
        STA TYPE,0        ; Read low, write high from end to beginning
        JSR COUNTDN

        LDY #1
        LDA #0
        STA TYPE,0        ; Read high, write nothing from beginning to end
        JSR COUNTUP

;----------------------------------------------------------------------

SEQ4    LDY #1
        LDA #1
        STA TYPE,0        ; Read high, write low from end to beginning
        JSR COUNTDN

        LDY #0
        LDA #0
        STA TYPE,0        ; Read low, write nothing from beginning to end
        JSR COUNTUP
```

F 5

```
;*****************************************************************************;
;
;       These next four sequences provide the coupling fault coverage.
;
;*****************************************************************************

SEQ5    LDY #0
        LDA #2            ; Read low, write high-low from beginning to end
        STA TYPE,0
        JSR COUNTUP

        LDY #0
        LDA #0
        STA TYPE,0        ; Read low, write nothing from end to beginning
        JSR COUNTDN
;-----------------------------------------------------------------------------

SEQ6    LDY #0
        LDA #2
        STA TYPE,0        ; Read low, write high-low from end to beginning
        JSR COUNTDN

        LDY #0
        LDA #0
        STA TYPE,0        ; Read low, write nothing from beginning to end
        JSR COUNTUP

;-----------------------------------------------------------------------------

SET     LDX #0FFH
        JSR INIT          ; Set all of test memory high

;-----------------------------------------------------------------------------

SEQ7    LDY #1
        LDA #2            ; Read high, write low-high from beginning to end
        STA TYPE,0
        JSR COUNTUP

        LDY #1
        LDA #0
        STA TYPE,0        ; Read high, write nothing from end to beginning
        JSR COUNTDN

;-----------------------------------------------------------------------------

SEQ8    LDY #1
        LDA #2
        STA TYPE,0        ; Read high, write low-high from end to beginning
        JSR COUNTDN

        LDY #1
        LDA #0
        STA TYPE,0        ; Read high, write nothing from beginning to end
        JSR COUNTUP
```

F 6

```
;**********************************************************************
;
;          This is the end of the 8 step memory test procedure.  Now some
;          housekeeping must be done, such as updating the error buffer if
;          a different number of errors were found this time, incrementing
;          the pass counter, etc.
;
;**********************************************************************
          INC PASS,0
          BNE FINISH1      ; Increment counter containing number of times
          INC PASS+1,0     ; memory test has been executed

;         Check current error count to error count from last pass

FINISH1 LDA LASTPAS,0
          CMP ERCOUNT,0    ; Compare number of errors detected on this pass
          BNE FINISH2      ; to number detected during last pass

          LDA LASTPAS+1,0
          CMP ERCOUNT+1,0 ; If different, update the error buffer
          BEQ FINISH3

;         If different number of errors, update error buffer and send new
;         values to screen

FINISH2 LDX #3            ; Clear indexing offset
          LDY #0

UPDATE  LDA ERCOUNT,X,0 ; Pick up number of errors detected on this pass, as
          STA (BUFFPT),Y  ; well as number of pass, and put these values to
          JSR DSPLY1M     ; the error buffer and the screen
          INC BUFFPT,0
          DEX             ; Do for all 4 bytes
          BPL UPDATE

          JSR CRLF        ; Send out carriage return-line feed for viewing ease

          LDA BUFFPT,0    ; Get low address of next buffer location
          BNE FINISH4     ; If non-zero, no wrap-around yet

          INC BUFFPT+1,0  ; If low address zero, need to increment high address
          LDA BUFFPT+1,0
          CMP #ENDBUF     ; Test to see of we are at end of available buffer
          BNE FINISH4     ; If at end of buffer, quit
          JMP OPEN
```

F 7

```
;        Since we are done with this pass, update last pass error count and
;        zero current error count

FINISH4 LDA ERCOUNT,0
        STA LASTPAS,0     ; Transfer low byte
        LDA ERCOUNT+1,0
        STA LASTPAS+1,0   ; Transfer high byte

FINISH3 LDA #0
        STA ERCOUNT,0     ; Clear the error counter
        STA ERCOUNT+1,0

        LDA PIA1BC
        EOR #8            ; This toggles the LED to indicate test is still
        STA PIA1BC        ; going

        JMP WAIT          ; Go and execute the test again
```

```
;   :::::::::::::::::::::::::::   SUBROUTINES.   ::::::::::::::::::::::::::::::::
;
;*********************************************************************************
;
;                       Initialize Test Memory Subroutine
;
;       This subroutine will set the entire test memory to whatever value is
;       sent to it in the X reg.  It will be used twice, once to clear the
;       test memory and once to set the entire memory high.
;
;       Regs Destroyed : A
;
;*********************************************************************************

INIT        LDA #0B0H
            STA PIA2AP          ; Initialize the address pointers
            LDA #0
            STA PIA1AP

            LDA #4
            STA ADCOUNT,0       ; We will do 1024 locations

INIT1       STX PIA1BP          ; store char out to port

            LDA PIA2AP          ; pick up control lines
            AND #1FH            ; drop all three
            STA PIA2AP          ; send them to RAM

            ORA #0B0H           ; set control lines back high
            STA PIA2AP          ; send them to RAM

            INC PIA1AP          ; Increment next address counter
            BNE INIT1           ; If less than 255, continue

            INC PIA2AP          ; If greater than 255, increment high address counter
            DEC ADCOUNT,0       ; Check to see if we are done with 1K
            BNE INIT1           ; If not, continue

            RTS                 ; Return to those who invoked us
```

```
;***********************************************************************
;
;                    Traverse Test Memory Subroutine
;
;      This subroutine does all the work.  Here we traverse the memory
;      either up or down and do all the reads and writes desired by the
;      calling routine.
;
;      The Y reg contains the type of value a read expects to find.  If Y
;      contains a 0, the routine expects a read to return a low value, and if
;      the Y reg contains a 1, a high value is expected.  If a value other
;      than the expected one is returned, an error is indicated.
;
;      The memory word TYPE contains the number of writes to be performed
;      following the read operation.  It can take on the values 0, 1, and 2,
;      corresponding to no writes, one write or write-write.  The value
;      written by the subroutine is always the opposite of what it read
;      from the memory location.  In the case of the two writes, in inverts
;      twice, so the memory location winds up containing what it started
;      with.
;
;      The memory word DIRECTION contains the mode of memory traversal.  If
;      DIRECTION is zero, we are traversing memory from the highest address
;      to the lowest, and if DIRECTION is non-zero we are traversing from
;      the bottom up.
;
;***********************************************************************
;
;           Set up necessary conditions for beginning to end traversal

COUNTUP LDA #0          ; If we are counting up, we will want to
        STA PIA1AP      ; begin at the beginning of the test memory
        LDA #0B0H       ; and go until the end
        STA PIA2AP      ; Set up high address lines and control lines
        STA DIRECT,0    ; Non-zero is count up
        BNE TESTPRO

;           Set up necessary conditions for end to beginning traversal

COUNTDN LDA #0FFH       ; If counting down, begin at end of test memory
        STA PIA1AP      ; and count down
        LDA #0B3H
        STA PIA2AP      ; Set up high address lines and control lines
        LDA #0
        STA DIRECT,0    ; Zero is count down
```

F 10

```
;          This is the actual traversal code

TESTPRO  LDA #4
         STA ADCOUNT,0      ; We will do 1024 locations

UP1      LDA #80H           ; Set the high bit in the memory word
         STA MASK,0         ; This tells us which bit we are looking at
         LDA #4             ; We are going to do the 4 bits in the memory word
         STA SHFTCNT,0      ; from left to right.

;          This is the inner loop that continually gets the word from the test
;          memory and increments the error counter based on expected values vs.
;          returned values

GET      LDX PIA1BP         ; get character
         TXA                ; Move it to A reg

         CPY #0             ; See if we are expecting a high or low value
         BEQ READLOW        ; If low, go to part that expects low value

         AND MASK,0
         BNE NEXT           ; Check to see value came back high as we expected
         BEQ BAD            ; If not, increment error counter

READLOW  AND MASK,0         ; We should read a low value
         BEQ NEXT           ; If we do, no error

BAD      TXA                ; Move the value we read into the A register so we
         EOR MASK,0         ; can set the bit to the value it was supposed to
         TAX                ; be so that writes will write the right value

         INC ERCOUNT,0
         BNE NEXT           ; If there was an error, increment the error
         INC ERCOUNT+1,0    ; counter for this pass

NEXT     LDA TYPE,0         ; Do we want to write back out?
         BEQ DONE           ; If not, do not

         TXA                ; Move what we got back into A so we can manipulate it
         EOR MASK,0         ; Change state of selected bit
         TAX                ; Put it back in X in case we want a second write
         LSR
         LSR
         LSR                ; Move it down into lower nibble
         LSR

         STA PIA1BP         ; store char out to port

         LDA PIA2AP         ; pick up control lines
         AND #1FH           ; drop all three
         STA PIA2AP         ; send them to RAM

         ORA #0B0H          ; set control lines back high
         STA PIA2AP         ; send them to RAM
```

F 11

```
        LDA TYPE,0          ; Do we want to do two writes?
        CMP #2              ; If not, do not
        BNE DONE

        TXA                 ; Move what we got back into A so we can manipulate it
        EOR MASK,0          ; Change state of selected bit
        LSR
        LSR
        LSR                 ; Move it down into lower nibble
        LSR

        STA PIA1BP          ; store char out to port

        LDA PIA2AP          ; pick up control lines
        AND #1FH            ; drop all three
        STA PIA2AP          ; send them to RAM

        ORA #0B0H           ; set control lines back high
        STA PIA2AP          ; send them to RAM

;       Here we have finished working with a bit and are ready to proceed
;       to the next one

DONE    LSR MASK,0          ; Shift bit it mask one place to right
        DEC SHFTCNT,0       ; Count of how many times we have shifted it
        BNE GET             ; We want a total of 4 shifts within same word

;       Here we have finished with one word and are ready to start on a
;       new word

        LDA DIRECT,0        ; Check for direction of memory traversal
        BNE UP              ; Non-zero represents upward

        DEC PIA1AP          ; Decrementing through memory, decrement
        LDA PIA1AP          ; address counters
        CMP #0FFH           ; If wrap-around, decrement upper byte
        BEQ NEXT1
        JMP UP1             ; In-line code has become to long for simple branches

NEXT1   DEC PIA2AP
        DEC ADCOUNT,0       ; Check for completion of traversal
        BEQ BACK
        JMP UP1             ; Not done, do again

UP      INC PIA1AP          ; Increment next address counter
        BEQ UP3
        JMP UP1             ; If less than 255, continue

UP3     INC PIA2AP          ; If greater than 255, increment high address counter
        DEC ADCOUNT,0       ; Check to see if we are done with 1K
        BEQ BACK            ; If done, exit
        JMP UP1             ; If not, continue

BACK    RTS
```

```
;*********************************************************************
;
;        The following routines are only for doing terminal I/O and have
;        nothing at all to do with the actual memory testing program
;
;*********************************************************************
;*********************************************************************
;
;                        Get starting time
;
;        This subroutine gets four characters from the serial device and
;        stores them as the starting time of the run
;
;        Regs Destroyed : X,Y,A
;
;*********************************************************************

GETIME: LDX #3              ; Set counter

TIM2:   JSR ECHOM           ; Get character from serial device and echo it

        STA TIME,X,0        ; Store it in beginning time location
        DEX                 ; Do until done
        BPL TIM2

        RTS                 ; Go back




;*********************************************************************
;
;                        Send Byte as Two Ascii Chars
;
;        This subroutine takes the byte in the A register and transmits it as
;        two ascii characters to the serial device.
;
;        Regs Destroyed : X,A
;
;*********************************************************************

DSPLY1M PHA                 ; Save the value on the stack
        LSR
        LSR                 ; Move it into lower nibble
        LSR
        LSR
        JSR ASKIZM          ; Asciiize the lower nibble (this also send the char)

        PLA                 ; Get the original value back from the stack
        AND #0FH            ; Clear off the top nibble
        JSR ASKIZM          ; Asciiize the low nibble and send char again

        RTS
```

F 13

```
;**********************************************************************
;
;                  Get Character from UART and echo it
;
;     This subroutine gets a character from the serial device, echos it,
;     and sends it back out to the UART so the sender knows it was received.
;
;     Regs Destroyed : A
;
;**********************************************************************

ECHOM:    LDA CUARTM
          AND #1            ; Wait for incoming character
          BEQ ECHOM

          LDA DUARTM        ; Load Character
          AND #7FH          ; Clear off parity bit
          PHA               ; Save it on stack
          JMP SENDM+1       ; Send it back as it came in




;**********************************************************************
;
;                  Convert byte into a valid ascii character
;
;     This subroutine converts the value stored in the A register into
;     a valid ascii character.
;
;     Regs Destroyed : A
;
;**********************************************************************

ASKIZM:   ORA #30H          ; Set Ascii bit

          CMP #3AH          ; Check for letter
          BMI SENDM         ; If not, send it as it is

          CLC               ; If it is a letter, we must make it a valid ascii
          ADC #07           ; letter
```

F 14

```
;********************************************************************
;
;                   Send character to serial device
;
;     This routine sends out whatever is in the A reg to the UART.
;     It waits (unnecessarily?) for the UART to finish the last
;     transmission before starting a new one.
;
;     Regs Destroyed : NONE
;
;********************************************************************

SENDM:  PHA                    ; Save the A reg on the stack

        LDA CUARTM
        AND #2                 ; Wait until transmit buffer is empty
        BEQ SENDM+1

        PLA                    ; Get character to be sent back from the stack
        STA DUARTM             ; Send the char

        RTS




;********************************************************************
;
;                   Send Carriage return - Line feed
;
;     For ease of viewing, many CR-LF combinations must be sent.  In order
;     to make the code more readable, this subroutine has been added.  It
;     performs the transmission of the CR-LF pair.
;
;     Regs Destroyed : A
;
;********************************************************************

CRLF    LDA #0DH               ; Load up the carriage return value
        JSR SENDM              ; Send it to serial device

        LDA #0AH               ; Load the line feed value
        JSR SENDM              ; Send it to serial device

        RTS                    ; All done
```

F 15

```
;*******************************************************************************
;
;                        Print Error Buffer Program
;
;       This program is essentially autonomous in that it is not needed to
;       execute the memory test program.  However, it is included with the
;       memory test program since it uses some of the same variables and
;       is used to print out the results of the memory test.
;
;*******************************************************************************

PRNTBF  EQU 00              ; Location used by the print routine

        ORG OFFOOH          ; start us at new known location

RESULT  LDA #BEGBUF
        STA PRNTBF+1,0      ; Initialize the print routines memory locations
        LDA #0             ; to the same ones the memory test used
        STA PRNTBF,0       ; (point us at the start of the error buffer)

        JSR CRLF           ; Send carriage return-line feed

        LDX #3
TMEOUT  LDA TIME,X,0
        JSR SENDM          ; The first thing sent out by this print routine is
        DEX                ; the time the memory test was started.
        BPL TMEOUT

        JSR CRLF

        LDY #4
COMPAR  LDA PRNTBF,0       ; See if the next location to be printed is the same
        CMP BUFFPT,0       ; as the next open slot in the error buffer.  If so,
        BNE OUTER          ; we may be done printing

        LDA PRNTBF+1,0
        CMP BUFFPT+1,0     ; Compare high bytes of the address.  If equal, we are
        BEQ QUIT           ; definitely done printing

OUTER   LDX #0
        LDA (PRNTBF,X)     ; Pick up value stored at location pointed at by
        JSR DSPLY1M        ; PRNTBF and send it to serial device
        DEY
        BNE OUTER2         ; Do for the four characters to be sent

        JSR CRLF           ; Send for appearance sake
        LDY #4

OUTER2  INC PRNTBF,0       ; Move print buffer pointer to the next location
        BNE COMPAR         ; in the error buffer

        INC PRNTBF+1,0     ; Increment the high order address byte of the
        BNE COMPAR

QUIT    JMP OPEN           ; when all done printing, quit
```

F 16

```
;*******************************************************************************
;       End of print results program.  All that remains is to set up
;       interrupts.
;*******************************************************************************

        RST OPEN            ; ON RESET, GO TO INITIALIZE STUFF

        IRQ BREAK           ; ON SOFTWARE INTERRUPT, GO HERE

        END
```

MEMORY TEST ERROR CHECKING ROUTINE

Return from Subroutine

T- MEMORY TEST PROGRAM

Char=T ?

Initialize Time ect.

Wait for 5 second Sync Signal

Do Memory Test Sequences 1-8

No. of Errors On Last Pass = Errors on Present Pass

Update Error Buffer

Display Pass Number & No. of Errors

Increment Buffer Pointer

Set No. of Errs Last Pass = No. of Errs in Present Pass

End of Buffer

Clear Present Pass Err Counter

Return to Monitor

Toggle LED To indicate Test cont.

P-PRINT RESULTS OF MEMORY TEST

```
   ____
  /    \              _____            ┌──────────────┐
 / Char=P_____/           \───▶       │  Get Error   │
 \    ?  /           _____/            │ Buffer Pntr  │
  \____/                                      │   Address    │
                                              └──────┬───────┘
                                                     │
                                                     ▼
                                              ⎛   Display    ⎞
                                              ⎜  Experiment  ⎟
                                              ⎝  Start time  ⎠
                                                     │
                                                     ▼
                                                  ____
                                                 /    \        1
                                ┌───────────────/ Done \──────────▶   Return to Monitor
                                │               \   ?  /
                                │                \____/
                                │                   │ 0
                                │                   ▼
                                │            ┌──────────────┐
                                │            │     Get      │
                                │            │   Contents   │
                                │            │  of Address  │
                                │            └──────┬───────┘
                                │                   ▼
                                │            ⎛   Display    ⎞
                                │            ⎝   Contents   ⎠
                                │                   │
                                │                   ▼
                                │            ┌──────────────┐
                                │            │  Increment   │
                                │            │    Buffer    │
                                │            │    Memory    │
                                │            │   Address    │
                                │            └──────┬───────┘
                                │                   │
                                └◀──────────────────┘
```

G3

MEMORY MAP

CHIP SELECTS	ADDRESS BLOCK SELECTED	SYSTEM DEVICE ENABLED
CS31	0F800H - 0FFFFH	SYSTEM EPROM
CS30	0F000H - 0F7FFH	
CS29	0E800H - 0EFFFH	
CS28	0E000H - 0E7FFH	6840 PTM
CS27	0D800H - 0DFFFH	6821 PIA #1
CS26	0D000H - 0D7FFH	6821 PIA #2
CS25	0C800H - 0CFFFH	6850 UART (SPARE)
CS24	0C000H - 0C7FFH	6850 UART
CS23	0B800H - 0BFFFH	
CS22	0B000H - 0B7FFH	
CS21	0A800H - 0AFFFH	
CS20	0A000H - 0A7FFH	
CS19	09800H - 09FFFH	
CS18	09000H - 097FFH	
CS17	08800H - 08FFFH	
CS16	08000H - 087FFH	
CS15	07800H - 07FFFH	
CS14	07000H - 077FFH	
CS13	06800H - 06FFFH	
CS12	06000H - 067FFH	
CS11	05800H - 05FFFH	
CS10	05000H - 057FFH	
CS09	04800H - 04FFFH	
CS08	04000H - 047FFH	
CS07	03800H - 03FFFH	
CS06	03000H - 037FFH	
CS05	02800H - 02FFFH	
CS04	02000H - 027FFH	
CS03	01800H - 01FFFH	
CS02	01000H - 017FFH	
CS01	00800H - 00FFFH	SYSTEM RAM
CS00	00000H - 007FFH	SYSTEM RAM

All select lines produced by the two 74154 4-16 decoders

CPU AND DECODER LOGIC

MEMORY

SERIAL I/O

H4

390Ω

+5

+5

19 CB2 Vcc 20

34 RESET
36 RS0
35 RS1
21 R/W
22 CS0 +5
24 CS1
23 CS2

26 D7
27 D6
28 D5
29 D4
30 D3
31 D2
32 D1
33 D0

6821

PA0 2
PA1 3
PA2 4
PA3 5
PA4 6
PA5 7
PA6 8
PA7 9

PB0 10
PB1 11
PB2 12
PB3 13
PB4 14
PB5 15
PB6 16
PB7 17

E 25 GND 1

E 25 GND 1

E1 2   E2 19

244

2 18
4 16
6 14
8 12
11 9
13 7
15 5
17 3

Vcc GND

+5

Vcc GND

244

2 18
4 16
6 14
8 12
11 9
13 7
15 5
17 3

E2 17   E1 1

150Ω

150Ω

Connector to
RMUT

① A0
② A1
③ A2
④ A3
⑤ A4
⑥ A5
⑦ A6
⑧ A7

⑨ A8
⑩ A9
⑪ WE
⑫ CS
⑬ D0
⑭ D1
⑮ D2
⑯ D3
⑰ GND
⑱ Vcc

26 D7
27 D6
28 D5
29 D4
30 D3
31 D2
32 D1
33 D0

6821

E GND

PA0 2
PA1 3
PA2 4
PA3 5
PA4 6
PA5 7
PA6 8
PA7 9

23 CS2
22 CS0
24 CS1 +5
21 R/W
55 RS1
36 RS0
34 RESET

CB1 Vcc
18 20

+5

RMUT INTERFACE

H5